DOCUMENT RESUME

ED 178 060 IR 007 802

AUTHOR	Hayes-Roth, Frederick; And Others
TITLE	Machine Methods for Acquiring, Learning, and Applying Knowledge.
INSTITUTION	Rand Corp., Santa Monica, Calif.
REPORT NO	RAND-P-6241
PUB DATE	Oct 78
NOTE	84p.
AVAILABLE FROM	The Rand Corporation, 1700 Main Street, Santa Monica, California 90406 ($5.00 per copy)
EDRS PRICE	MF01 Plus Postage. PC Not Available from EDRS.
DESCRIPTORS	*Artificial Intelligence; Automation; *Computer Oriented Programs; *Cybernetics; Game Theory; *Information Processing; *Programing
IDENTIFIERS	Expert Knowledge

ABSTRACT

A research plan for identifying and acting upon constraints that impede the development of knowledge-based intelligent systems is described. The two primary problems identified are knowledge programming, the task of which is to create an intelligent system that does what an expert says it should, and learning, the problem requiring the criticizing and expanding of current knowledge to improve overall system performance. In this view, learning produces new knowledge which must be accommodated to implement an improved system. Motivating problems, technical requirements for supporting system extensibility and improvability, and research proposed to achieve these objectives are described and related to previous research. Numerous examples are drawn for a heuristic program that plays a moderately difficult card game (hearts). Several appendices provide additional details on many technical issues, including the representation of knowledge and the structure of knowledge bases, the design of a knowledge programmer, various control methods, the design of a learning workbench, an illustrative learning scenario, and various learning heuristics. (Author)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED178060

U.S. DEPARTMENT OF HEALTH, EDUCATION & WELFARE
NATIONAL INSTITUTE OF EDUCATION

THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

MACHINE METHODS FOR ACQUIRING, LEARNING, AND APPLYING KNOWLEDGE

Frederick Hayes-Roth, Philip Klahr,

John Burge, David Jack Mostow

October 1978

"PERMISSION TO REPRODUCE THIS MATERIAL IN MICROFICHE ONLY HAS BEEN GRANTED BY
Rand Corp.

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

P-6241

IR007802

The Rand Paper Series

Papers are issued by The Rand Corporation as a service to its professional staff. Their purpose is to facilitate the exchange of ideas among those who share the author's research interests; Papers are not reports prepared in fulfillment of Rand's contracts or grants. Views expressed in a Paper are the author's own, and are not necessarily shared by Rand or its research sponsors.

The Rand Corporation
Santa Monica, California 90406

ABSTRACT

Recent advances in intelligent systems have emphasized the use of "expert knowledge" to solve problems. This paper describes a research plan for attacking the core problems impeding the rapid development of such knowledge-based systems. We identify the two chief problems as knowledge programming and learning. The task of knowledge programming is to create an intelligent system that does what an expert says it should. The learning problem requires criticizing and expanding current knowledge to improve system performance. In this view, learning produces new knowledge which must be accommodated to implement an improved system. This accommodation requires a capability for incremental knowledge programming. After describing the motivating problem in detail, we define the technical requirements for supporting system extensibility and improvability. Research proposed to achieve these objectives is described and related to previous work. Numerous examples are drawn from a heuristic program that plays a moderately difficult card game (hearts). Several appendices provide additional details on many technical issues, including: the representation of knowledge and the structure of knowledge bases; the design of a knowledge programmer; various control methods, including caching and demons; the design of a learning workbench; an illustrative learning scenario; and various learning heuristics.

CONTENTS

PREFACE iii

ABSTRACT v

1. OVERVIEW 1

2. THE PROBLEM: DEVELOPING AND IMPROVING INTELLIGENT AGENTS RAPIDLY 3

3. TECHNICAL PROBLEMS IMPEDING KNOWLEGE PROGRAMMING AND LEARNING 6

4. THE PROPOSED APPROACH 10

5. RELATED PREVIOUS RESEARCH 31

REFERENCES 37

APPENDIX I:
CONCEPTUAL REPRESENTATION OF HEURISTICS 43

APPENDIX II:
THE DESIGN OF THE KNOWLEDGE PROGRAMMER 52

APPENDIX III:
CONTROL METHODS 56

APPENDIX IV:
CACHING AND DEMONS 59

APPENDIX V:
DESIGN FOR A LEARNING WORKBENCH 63

APPENDIX VI:
A LEARNING SCENARIO 66

APPENDIX VII:
LEARNING HEURISTICS 70

APPENDIX VIII:
RULES OF HEARTS AND SAMPLE INITIAL HEURISTICS SET 75

APPENDIX IX:
CRITERIA FOR A WELL-FORMED KNOWLEDGE BASE FOR INTELLIGENT AGENTS 77

1. OVERVIEW

Recent advances in intelligent systems have depended primarily on the capacity to apply "expert knowledge" to limited problems within a specific domain. As a result of the new emphasis on knowledge-based systems (KBSs), several practical and significant intelligent systems have been fielded. As a consequence of these successes, the current demand for additional applied artificial intelligent (AI) systems vastly exceeds the potential supply. The disparity between required and available resources will probably increase in the next 20 years.

The principal bottlenecks in developing KBSs are acquiring and implementing domain-specific problem solving knowledge. With current methods, someone who knows how to solve problems manually must work intimately with an AI programmer to recreate the methods in a computational device. Three major obstacles impede the economical development of the desired system: formulating the necessary knowledge; implementing it; and discovering new knowledge to improve the initial implementation. Each of these is discussed in turn.

The first difficulty arises because the expert finds it difficult to communicate with the programmer. Each party conceives of the problem in terms of different conceptual domains. The expert understands his problem in terms of relevant domain concepts, such as "risk," "scenarios," "plans," "intentions," "disease," "symptoms," etc. He formulates his problem solving strategies in terms of heuristic methods that operate upon and transform these types of problem components. When asked to operationalize these concepts and methods in terms of program entities familiar to the AI programmer, the expert is literally stymied. Conversely, only a highly-trained (often Ph.D.-level) AI programmer can convert these ideas into problems amenable to familiar inference mechanisms.

The second problem arises when the programmer tries to implement the problem solving knowledge. Very few types of knowledge seem to fit neatly into existing schematizations. Implementing the knowledge requires handcrafting code (usually in LISP) and making numerous arbitrary decisions to integrate diverse kinds of knowledge. The complete implementation, even for simple intelligent agents, seems to take too long. Few simple tasks are solved without significant programming expense.

And, when the program finally runs, it inevitably exhibits shortcomings, oversights, and errors. This is the source of the third problem. At this point, the project team moves from its initial task of "knowledge programming" to a new task of "knowledge discovery" or "learning." Diagnosing and fixing most of the weaknesses in the initial implementation require new interactions between the domain expert and programmer. During this phase, too often they discover that the new knowledge cannot easily be integrated into the current system design. Either the program is reimplemented or much of the newly developed knowledge, which seems too costly to translate into new capabilities, is sacrificed.

In this paper, we develop a research plan for attacking the core problems impeding the rapid development of extensible intelligent agents. As suggested by the preceding discussion, we identify the two principal subtasks in intelligent system development as knowledge programming and learning. We plan to develop computer aids to automate or assist in the manual performance of these tasks. The task faced in knowledge programming is creating an intelligent agent that does what an expert says it should. The task faced in learning is criticizing and expanding current knowledge to improve performance. In this paper, we will formulate specific capabilities for two intelligent systems, a Knowledge Programmer (KP) and a Learner, that we intend to develop to perform these tasks.

The technical problems we identify include the following. First, we need a capability for accepting and understanding descriptions of entities, problems, and heuristic methods in an arbitrary domain of interest. Once we have understood what needs to be done, we have to implement these descriptions. Implementation requires fleshing out informal aspects of the domain descriptions and operationalizing the method interpretations. Once an initial implementation executes, learning requires heuristic methods for detecting weaknesses, reinforcing strengths, and capitalizing on other opportunities for program enhancement. Often, we must conjecture likely but not necessarily certain ways to improve behavior. Then these new ideas must be reimplemented and tested in a working system.

Our approach to these problems is to implement intelligent systems that provide the needed capabilities. Specifically, we have developed methods for understanding the kind of domain knowledge that a domain expert would like to communicate to a programmer. Once the domain concepts and methods have been understood, other mechanisms transform the declarative knowledge into an executable form, called an intelligent agent (IA). This transformation requires operationalizing the transactional aspects implicit in the agent's description and fitting them into one or more problem solving frameworks in the programmer's repertoire. Then traces of the agent's performance are collected, and other heuristic methods detect problems and conjecture solutions. The conjectures include new domain concepts, problem solving heuristics, and implementation suggestions. These conjectures, combined with previous but still valid components of the knowledge base, constitute a new set of expert conceptions regarding the task. These are reimplemented, and the iterative learning cycle repeats.

The remainder of this paper comprises the following sections. Section 2 describes the motivating problem in more detail. Section 3 relates our stated goals of extensibility and improvability through learning to several required technical capabilities. Section 4 describes our proposed approach toward providing these capabilities. Section 5 describes relevant prior research by ourselves and by others. Several appendices provide additional information on many of the technical problems at a finer level of detail than that found in the body of this paper.

2. THE PROBLEM: DEVELOPING AND IMPROVING INTELLIGENT AGENTS RAPIDLY

Intelligent agents (IAs) are programs that solve simple to moderately complex problems using knowledge of the problem domain and heuristic problem solving methods. This informal definition captures the primary features of what people colloquially mean by "heuristic programs" or "knowledge-based systems." We envisage a day when many mundane tasks will be performed for people by numerous computer-based specialists, each an IA in a specific domain.

To bring that day closer, we need to increase significantly the "manufacturing" capacity of the AI field. One way would be to train more AI programmers. That effort will not significantly alter the critical shortage of programming resources, primarily because the increasing demand for agents will exceed considerably the increasing supply of AI programmers. The second method would be to develop intelligent systems that can automate much of the work required to develop agents.

We perceive the key to success in this effort to lie in finding a way to make heuristic programming tools that can exploit the ability of an expert to explain, in terms of his own high-level domain concepts, what his IA should do. Thus, we distinguish our goal from that of typical automatic programming (AP). In current AP research, the primary goal is to convert formally specified input-output requirements into verifiable code. The emphasis on automatically discovering the means of achieving some desired result is significantly different than our own. In our task, we expect an expert to tell us both <u>what</u> to do and <u>how</u> to do it. When he tells us what to do, he specifies what the inputs and outputs must be and any other constraints that must be satisfied by the agent. But when he tells us how to do the task, he advises us as best he can what heuristics and problem solving strategies to employ. All that remains for the Knowledge Programmer is to achieve some feasible realization of his specifications.

If an effective KP can be developed, this approach might prove to be both more powerful and more generally applicable than AP. Because an AP system must understand a problem, generate an algorithm, develop a software design, and code a program to achieve it, such a system must be very intelligent. As a consequence, the magnitude of AP problems that we should anticipate solving in the foreseeable future is necessarily limited. AP simply requires too much intelligence. In our approach however, we assume that an expert already understands the problem and has effective methods for solving it. All the KP has to do is accept his definitions and find a feasible way to perform them computationally. This itself is a difficult task but much less so than the general AP problem.

The KP's task is difficult for several reasons. First, by allowing an expert to state his requirements in terms of concepts natural to his own domain (in order to facilitate his task of communicating knowledge), we open the possibility of having to formalize his informal notions. Second, many suggested methods are incomplete. The KP must choose reasonable ways to make the expert's

abstract ideas concrete. Third, in most human tasks, heuristic methods vary widely in their domains of applicability, intended functions, and certainty. Moreover, in many cases, a set of suggested problem solving approaches will be inconsistent, in the sense that pursuing one approach precludes simultaneously pursuing others.

Many of these problems have been avoided in previous IA development projects by narrowly restricting all knowledge to one particular format that could support a single type of deduction. For example, the rule-based systems have primarily been applied to signal or symptom interpretation problems, where each rule relates antecedents to possible consequences. In such a problem, a simple uniform representation of knowledge with a trivial control program (e.g., backward-chaining) suffices. As soon as other task factors are introduced, these simple structures prove inadequate. For example, if the agent must decide how to allocate its own limited resources to collect useful data and generate appropriate actions, conventional approaches would require developing a complex, special-purpose program. Thus, although some popular current architectures have evolved to facilitate the development of IAs, they have very limited flexibility and do little to expedite the transfer of wisdom about how to solve problems into problem solving architectures and control structures.

We believe a significant need exists to develop mechanisms that could expedite the development and iterative improvement of knowledge-based IAs. For many problems of significant importance, the use of IAs will depend on their ability to be inspected and modified very rapidly. Conversely, if the only way AI programs can be modified is to send them back to the AI software shop for repairs, they will never be adopted for many of the tasks to which they are best suited. This judgment derives from the observation that IAs are uniquely well suited to performing in numerous, diverse decisionmaking situations of limited complexity. The criteria by which judgments will be accepted in these tasks will vary widely and change often. Whatever implicit assumptions are built into an IA by the programmer must be subject to rapid evaluation and modification. Finally, when the IA exhibits undesirable behavior, either because its methods are faulty or because the environment in which it operates has changed, it must be possible to modify it locally and without great expense. We can think of numerous application areas where IAs are already practical, except for the fact that they cannot currently be extended or modified easily. Among these we include military planning aids, secure message handlers, appointment assistants, command and control assistants, scheduling agents, simulated game opponents, traffic controllers, etc.

We also believe that the pursuit of KP and learning capabilities is encouraged by technological trends. Among the significant trends, the first seems to be the expanding number of computer systems. More people will be exposed to more computers than ever before, and the primary constraint limiting computer exploitation will be the inability of the R&D community to produce enough useful software. The second trend is the geometrically increasing ratio of programmer cost to computing costs. This trend strongly motivates efforts to capitalize on human judgment rapidly, even if the eventual software

product that results is less efficient than a handcrafted program. The third trend we note is the development of performance programs that depend on empirically acquired knowledge. By their very nature, knowledge-based programs are likely to require frequent modification and revision. This militates for the development of systems to assist in the formulation of new knowledge and for its automatic incorporation and integration into an existing IA.

Systems that can assist in the maintenance and improvement of knowledge bases would support numerous functions of national importance. In particular, we envisage man-machine systems playing increasingly important roles in governmental planning and control functions. The utility of conventional planning systems is severely limited by the attendant difficulties that occur when they require modification. For example, military mission planning packages incorporate numerous assumptions about resources, targets, and rules of engagement. As a consequence, the plans they produce are quite vulnerable to erroneous assumptions. This vulnerability could be reduced if planning knowledge bases could be revised appropriately in response to feedback obtained from plan outcomes. The requirements for that capability are a subset of those proposed for study in this project. Similar requirements for rapid and inexpensive means to assist in translating feedback into knowledge base modifications arise in most application domains we have considered.

3. TECHNICAL PROBLEMS IMPEDING KNOWLEDGE PROGRAMMING AND LEARNING

The goal of developing a learning capability (i.e., the ability to improve performance by incorporating new or improved knowledge) requires solutions to several technological problems. These problems arise from the general requirements of systems that learn by modifying knowledge. Some basic characteristics of such systems include the following. (1) Their knowledge bases must be extensible. (2) Knowledge accrues both from exogenous instruction and advice as well as from self-stimulated discovery. (3) Knowledge consists of different types of concepts, including category definitions, predicates, functions, heuristics, plans, etc. (4) Once a system begins to perform, eliminating behavioral problems can necessitate developing new knowledge of any of these types. (5) The same unit of knowledge can support many different types of actions. (6) Most units of knowledge refer to other elements of the knowledge base; thus knowledge is largely self-referential and hierarchically organized.

We view the overall problem of developing IAs as comprising two principal subtasks, knowledge programming and learning. In the remainder of this section, these are considered in turn.

3.1 KNOWLEDGE PROGRAMMING: DEVELOPING AN INITIAL CAPABILITY TO INCORPORATE AND DO WHAT YOU'RE TOLD

The development of an initial performance capability, in the proposed paradigm, requires a mechanism to translate instructions on "what to do" into some computationally interpretable representation. We see four primary problems in Knowledge Programming. First, we need to develop suitable vehicles to enable an expert to communicate his domain knowledge. Second, we need to define and implement the knowledge that an effective programmer would bring to bear in creating a program to apply the expert's knowledge. Third, to enable a user to follow its behavior, the resulting IA must be able to explain what it does in the conceptual terminology the expert used to create it. Finally, some mechanisms must be provided to enable the expert to examine and validate the acquired knowledge and its implementation. Each of these problems is discussed in turn.

3.1-1 Communicating Problem Domain Knowledge

To appreciate the problem of communicating problem domain knowledge, consider what you would need to tell a programmer who was assigned to you to create an IA. To be concrete, consider an IA intended to play the role of appointments secretary. The first thing you'd tell the programmer would be some basic definitions about the types of situations he needs to handle. For example, you might distinguish between different classes of people (e.g., superiors, subordinates, relatives, friends, athletic partners); different classes of meetings (e.g., project meetings, seminars, interviews, business and social luncheons); and different types of temporal periods (e.g., work days, weekends, meal times, holidays, athletic work-outs). You would also need to explain what the problem is and what would constitute problem solving actions. For example, you might say, "Assign periods of time to requests for appointments for workdays

that are not holidays when I intend to be in town." Of course, the programmer would then need to know how to determine holidays, travel plans, and so forth. Then you'd provide him some absolute rules, which should never be violated without explicitly raising exceptions. For example, you might say, "Always schedule requests from my relatives and superiors, and make sure I never go without eating lunch." Then, you would provide him with simple, conceptually high-level instructions about how to try to solve the scheduling problem. For example, you might say, "Try to arrange athletic dates before lunch," "Lunch must be before 3 pm and after 11:30 am," "Seminars are best around 10 am," "Requests from project leaders are more important than requests from subordinates," "Try to avoid meetings before 9 am, unless doing otherwise would delay meetings more than 3 work days," and "If a requested time is undesirable, suggest some preferable time to the requestor and ascertain whether it's acceptable to him."

The primary problems in communicating such knowledge to a programmer concern how to parse the natural language, how to relate the instructions to the meaningful domain concepts (e.g., holidays, superiors), and how to operationalize the absolute rules, and the suggested heuristics. These problems are complicated, as previously stated, by the fact that different heuristics address qualitatively different types of concerns.

3.1-2 Implementation Knowledge

Given this sort of task explanation, what knowledge would the programmer need to bring to bear to create the desired IA? The first type of decision he would probably make would concern the general structure of the program. For example, he might adopt an AI paradigm such as generate-and-test. Then, he could decide which rules would be good candidates for generating plausible hypotheses (tentative scheduling decisions). The other rules would be used to test the acceptability of these hypotheses.

The programmer's task after these initial high-level decisions would become considerably more mundane and aesthetically dissatisfying. He would uncover numerous decision points requiring arbitrary selections (e.g., "How much desirability should accrue to achieving each stated objective?" or "Which is more important, a meeting with x or a meeting with y?"). Moreover, he'd soon find that the hypothesize-and-test paradigm he previously adopted was not working out neatly. Many problems that the IA must solve don't directly concern appointment hypotheses. These other problems concern bookkeeping functions, how to collect additional information when required, maintaining tentative contexts to evaluate hypothesized interactions, merging incompatible suggestions, etc.

What typically results from these unanticipated problems is a program that, although it began with some neat overall design principles, ultimately incorporated numerous detailed and arbitrary implementation choices. Some of these produce desirable behavior, and others cause undesirable performance characteristics. Because the expert's specifications are both over-constrained (not all constraints can be satisfied in all cases) and under-constrained (many alternative

implementations would satisfy the same constraints), the programmer exercises enormous liberty in realizing the initial system. The freedom that comes from the large space of acceptable initial implementations reduces the requirements for rigor and wisdom on the part of the Knowledge Programmer.

3.1-3 Explaining the IA's Decisions

Ordinarily, the usefulness of an IA depends on its capacity to explain its decisions. Thus, the expert expects the programmer to provide output that relates the decisions to the initial conceptual specifications. If the programmer has had difficulty preserving the domain structure within his program, as is usually the case, this requirement is difficult to satisfy. To a large extent, the wide acceptance developed for the simple interpretation IAs that have employed backtracking rule-based systems derives from the fact that the implementations hardly obscure the expert's logic. On the other hand, most IAs will not fit into such a simple program structure, and the problem of explaining an implementation becomes more difficult.

3.1-4 Verifying the IA's Knowledge

In the next section, we discuss the empirical aspects of verifying knowledge. Before that, however, we ought to address the problem of allowing an expert to inspect the knowledge base constructed by the programmer. Again, unless the knowledge is of extremely limited types (such as hierarchies and production rules), this is very difficult. In typical IAs the knowledge becomes embedded in LISP procedures that are opaque to non-programmers. Furthermore, even if they are embellished with documentation describing their purpose, there is no guarantee that their procedural attachments do what is advertised. The more transparent knowledge representations, such as rules and semantic networks, seem much more satisfactory for these purposes. However, as previously noted, few if any IAs can be constructed in transparent ways from such components. Thus, it seems that more general capabilities are required to represent declaratively the diverse types of knowledge that go into an IA. This requirement, conversely, entails the need for more general application mechanisms that can execute declarative forms of knowledge.

3.2 LEARNING: DISCOVERING NEW KNOWLEDGE TO IMPROVE PERFORMANCE

As we know well, very few programs do what we want when they are first implemented. This holds for IAs as well. Even leaving aside the problem of bugs, because they're largely heuristic and based throughout on empirically derived knowledge, IAs frequently exhibit behaviors that we judge undesirable. Learning, in this context, means recognizing the occurrence of undesirable behaviors and identifying the knowledge sources that are responsible for them. We then modify prior knowledge in just those ways we believe will rectify the behavioral weaknesses. Often these modifications to prior knowledge are hypothetical conjectures. In these cases, the "proof of the pudding is in the eating." The knowledge is validated by reimplementing the IA and verifying that the previous problems have been eliminated.

A capacity for learning thus presupposes an ability to reimplement knowledge. This explains why the Knowledge Programmer is an essential component of our proposed learning research project. In addition to Knowledge Programming, learning requires several other capabilities. Spcifically, we must be able to collect data on actual behavior of the IA. Then, we must have an ability to identify undesirable behaviors. Each undesirable behavior must then be attributed to specific decisions, which in turn are attributable to specific prior knowledge. Creating new knowledge depends on an ability to conjecture changes to prior knowledge that would avoid those behaviors in the future. Finally, the learner must be able to evaluate experimentally his hypothesized new knowledge.

In addition to discovering new knowledge in response to undesirable outcomes, one can also attempt to formulate new knowledge to replicate intentionally those desirable outcomes that seemingly occurred by chance. Specifically, desirable consequences are attributed to decision outcomes that, in turn, can be attributed to particular knowledge. That knowledge is then modified accordingly to increase its generality and relative importance.

Moreover, one can justify a search for new knowledge based on weaker principles. For example, a learner could hypothesize new concepts and methods that vary previous knowledge in heuristically interesting ways. As long as these conjectures are considered tentative, there is likely to be no adverse consequences to this empirical approach to discovering new knowledge.

In all of these cases, methods are required to suggest new variants of old knowledge and to test their impact in new implementations.

4. THE PROPOSED APPROACH

This section describes the basic paradigm of our research and specific decisions we have tentatively made to address the technical problems previously identified. The first subsection provides an overview of the research project. Subsequent subsections explain our current choices for target applications and then our methods for approaching several core problems. Those discussed include: knowledge representation, advice taking, interpreting informality, operationalization of advice, control and efficiency considerations, learning and the iterative improvement of performance.

4.1 THE RESEARCH PARADIGM

The overall structure of the research project is shown in Fig. 1. The principal features of the research paradigm include: the acquisition of domain knowledge and problem solving suggestions from the expert; the merger of domain advice and knowledge of implementation techniques by KP to construct an initial IA; the collection and analysis of observed behavior by the Learner to conjecture new knowledge; and the iterative loop that allows the new knowledge to be implemented automatically within a new IA. More specifics on these functions will be provided later in this section and in the appendices.

Before considering our specific approaches, we need to identify a task domain suitable for illustrating our ideas. This is done in the next section.

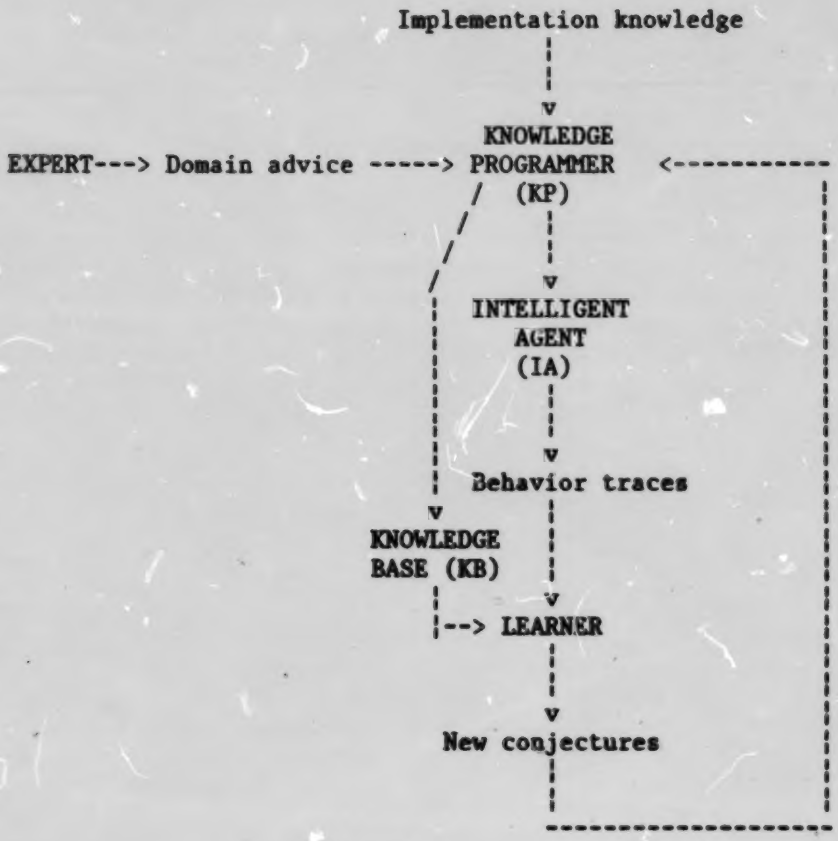

Figure 1. The research paradigm: The initial behavioral capability of the Intelligent Agent (IA) is generated from domain advice and implementation knowledge by the Knowledge Programmer (KP). The Learner examines the behavior traces of IA and the Knowledge Base (KB) created by KP to develop new conjectures. This hypothetical knowledge is reimplemented to test and evaluate it.

4.2 APPLICATION TASK DOMAINS

In this section, we discuss three issues. First, we attempt to formulate some criteria for identifying suitable task domains for applying our research. Second, we suggest several practical domains that seem to satisfy these criteria. Finally, we explain our reasons for choosing two specific tasks and introduce one of them that will be used to illustrate the research concepts throughout the remainder of the paper.

4.2-1 Criteria for IA Task Domains Suited to our Approach

The premise of the research is that many tasks can be performed by IAs of limited intrinsic complexity if a means can be found to

implement human advice about how to perform the tasks. In addition, we presume that automated aids for creating IAs are essential for economic reasons. How would we identify the class of tasks appropriate for this sort of approach? The chief characteristics of these tasks include the following. The tasks would require primarily symbolic inference, as opposed to numerical, statistical or signal processing. The knowledge required to perform the task would exhibit qualitatively diverse types, addressing different types of concerns. Despite its diversity, the expert would find it simple to formulate the knowledge. Notwithstanding its apparent simplicity, a programmer would find it difficult and tedious to implement that knowledge. Most of the methods suggested by the expert would be heuristic, rather than algorithmic. Truly good ideas about how to solve the problem would come ultimately from experimenting with different ideas and learning from experience.

4.2-2 Potential Practical Task Domains

Numerous practical tasks seem to satisfy all of these desiderata. Office automation tasks, such as billing, appointments, budgeting and filing, exhibit these properties--they are symbolic, heuristic, easily explained, constantly modified, and poorly implemented. More general resource allocation and scheduling problems fall within this category too. Space exploration vehicles, unmanned and investigating unfamiliar environments, would also exhibit the same properties. In this domain, more than any other, the need exists for rapidly modifiable heuristics concerned with numerous diverse issues. We can identify numerous other tasks with these properties in military mission planning and logistics, sophisticated medical instrumentation agents, and automated manufacturing. Another excellent domain of applicability includes intelligent partners, e.g., game players, simulated opponents, or tutors in narrow areas of expertise.

4.2-3 The Target Tasks: Hearts and Scheduling

We have tentatively chosen two tasks as immediate targets for applying our research. One is the card game Hearts and the other is office scheduling. Each of these is described briefly now.

The card game Hearts provides an excellent benchmark problem for refining and testing our ideas. The game exhibits all of the features of tasks appropriate to our approach. An experienced player can easily formulate the domain knowledge. The definitions of cards, players, points, tricks, rounds, games, etc., and the absolute rules of play are quite simple to express. Appendix VIII provides this short description for readers unfamiliar with the game. Most of the inferences required are symbolic, but a few involve numerical and statistical reasoning. A reasonably good initial capability can be explained in terms of a few heuristics for choosing cards and pursuing alternative strategies. Appendix VIII also lists such an initial set of heuristics created by an experienced player in roughly 5 minutes. Even such a simple initial description required several man-months to implement in INTERLISP, and the resulting implementation proved woefully inadequate to support extensions of additional knowledge.

Hearts also offers the potential of representing and using knowledge at different levels of abstraction and generality. The game involves methods of planning, of developing global strategies and local tactics. Hearts also facilitates the exploration of alternative control methods to apply knowledge (see Appendix III).

The Hearts domain exhibits many other properties which stimulate exciting research. Because the game involves several players, a single player must monitor the behavior of others to assess their intentions in order to choose appropriate strategies. Of course, most interpretations and intended actions are highly uncertain. There is no hope of "solving" the problem. Instead, heuristic means must be discovered to conjecture plausible interpretations, strategies and specific plays, and other heuristics must be employed to evaluate and accept or reject such suggestions. Even the question of deciding what information should be assessed requires heuristic approaches, since computing resources are limited in comparison with the set of potentially useful computations.

Finally, Hearts models in a simple way problems of international conflict and deterrence. In addition to its uncertain aspects, the game fosters spontaneous alliances of uncertain duration. Moreover, the game may be won either by minimizing the accumulation of small losses or by successfully risking and capturing all resources. In this regard, the game mimics the realities of international power politics, where two opposite strategies often suggest themselves. The first is to pursue peace by living with the opposition, with intermittent limited conflicts. The second is to pursue peace by initiating a successful all-out campaign to overwhelm the opposition. We believe that developing expert capabilities in this game could provide valuable insights into the problems of balancing and controlling such conflicting strategies. This seems a useful initial simplification of what is, of course, a significantly more complicated real-world problem.

In short, the task of developing an expert Hearts playing IA seems to satisfy all of the important desiderata for an applications task. Because it is both very constrained--largely symbolic, easily explained conflicting strategies--and highly under-constrained--enormous uncertainty, numerous possible implementations--Hearts is an excellent initial benchmark task.

In addition to the Hearts task, we propose to apply our KP and learning programs to a task in office scheduling. Specifically, we will test the generality of our methods by evaluating the system's capabilities to perform in the task domain of appointments scheduling. Typical aspects of this task were illustrated in Section 2. We believe that scheduling functions may well prove to be one of the largest domains of application of IAs. Because resource mixtures, objectives, strategies, tactics and opportunities change frequently, problems in this domain stress the timeliness requirements for modifiable IAs. Our initial problem in office scheduling would represent one arbitrarily selected element of this broad class of important functions.

Throughout the remainder of this paper, we illustrate our general scientific ideas with examples taken from the domain of Hearts. Because the rules of Hearts and the conceptual definitions are fixed, they provide unambiguous reference points. In addition, a complete initial set of heuristics suitable for developing an IA is quite small (see Appendix VIII). Nonetheless, the richness and complexity of the inferences required to accomplish the KP and learning tasks still overwhelm current AI capabilities.

The reader should realize we are aiming at the development of general methods for knowledge acquisition and learning. He should not get bogged down in the details of the Hearts model and examples. Appendix IX outlines several general criteria that characterize well-formed, extensible IA knowledge bases. The examples from the Hearts domain instantiate those abstract ideas in a concrete, easily interpreted fashion.

4.3 REPRESENTING KNOWLEDGE

In our development of knowledge representation schemes, we have emphasized declarative formalisms that could be evaluated or applied in various ways. We are developing representation schemes with formal semantics to clarify our interpretations of informal specifications. In addition, we have begun to formulate general mechanisms for applying the same knowledge unit (KU) in different ways to perform diverse functions.

Our representation scheme views each KU as a concept with various types of facets. The facets themselves may have different attributes, called aspects. Thus, our approach is similar to those pursued by other researchers in knowledge representation and semantic nets. Several things are novel about our work however. First, our knowledge structures have formal semantic properties. Second, we construct new structures dynamically as they are defined by exogenously provided expert advice. Third, we model nearly all components of an IA in terms of concepts, rather than restricting the application of knowledge structuring formalisms to simple sets and relations. Finally, we have developed representations for a very wide class of program entities, including predicates, functions, heuristics, plans and demons.

The general form of concept representation is illustrated in Fig. 2, taken from (Mostow, 1978).

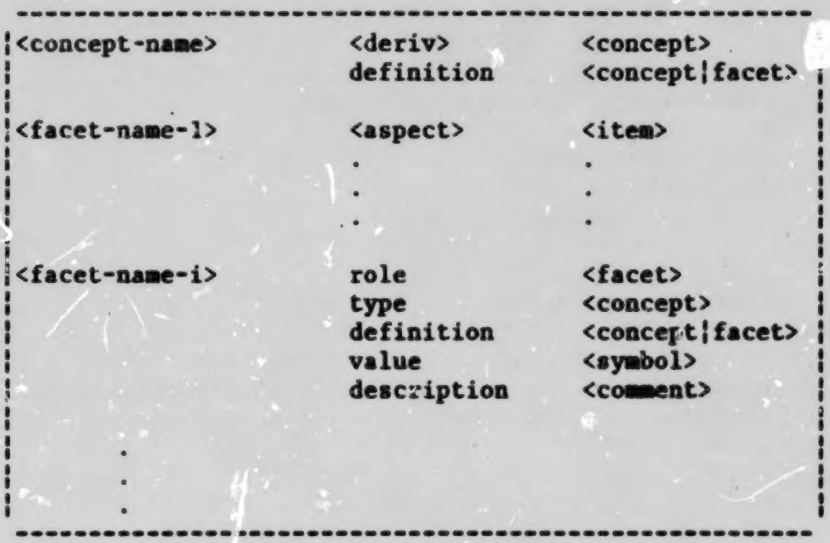

Figure 2: Representation of concepts

Each concept has a unique name, indicated by <concept-name> in the upper-left corner of the figure. A concept corresponds to a generalized lambda expression, and each facet of the concept corresponds to a typed lambda variable. The aspects of a facet describe its type, value, and so forth. The definition of a concept corresponds to the body of a lambda expression. It tells how to evaluate, apply or recognize instances of the concept. Of course, concepts can refer to other concepts.

Some specific examples will clarify the formalism. These examples are taken from a working program that was developed using the conceptual formalisms after an initial, handcrafted IA proved non-extensible. In the next section we will propose methods whereby such concepts can be created directly from expert advice. The first example illustrates the concept of the game rule that requires each player to follow the suit of the card led by the first player in a trick.

Example 1.

```
concept:  FOLLOWSUIT
        description:player must follow suit if possible
        type:       PLAYRULE
        returntype: SETOFCARDS
        context:    TRICKINPROGRESS
        arguments:
                name:   P
                type:   PLAYER
        locals:
                name:   S
                type:   SUIT
                value:  (SUITLED)
```

```
            name:       H
            type:       SETOFCARDS
            value:      (HAND P)
      necessary:        S
      definition:
            condition:  (VOID P S)
            value:      H
            condition:  otherwise
            value:      (CARDSINSUIT H S)
```

This concept is interpreted as follows. It is a rule of play which, when evaluated, will return a set of cards. It is always evaluated in the context of the trick in progress. It expects one argument (P) designating a particular player. It employs two local variables: S designates the suit of the card led to start the trick; H designates the set of cards that constitute the player's hand. S must be defined (non-nil), as it is after the first card in the trick is led. The possible cards to play include all of H if P is void in suit S, otherwise only the cards in suit S from the player's hand are permitted.

The second example illustrates a plan concept. This plan corresponds to the general strategy of trying to take a low profile, avoiding all situations where you might win tricks that include cards with point values (Hearts or the Queen of Spades). In this example and others we employ notational abbreviations to compact the descriptions.

Example 2.

```
concept   PLAY-LOW-PLAN
      type        PLAN
      context     TRICKINPROGRESS
      arguments   ((P PLAYER))
      when        (OR (TOOK OPPONENT POINTS)
                      (LACKING P HIGHCARDS)
                      (NOT (CONTROL P HEARTS)))
      why         (MINIMIZE (POINTS P))
      recommend   (((UNLOAD-POINTS P) EXCELLENT)
                   ((AVOID-POINTS P) VERYGOOD))
```

This concept takes one argument, namely the player P whose strategy might be to play low. The plan is relevant when some opponent has already taken points, P lacks very strong high cards, or when P does not control the hearts suit. Any of these conditions strongly indicates that taking all the points ("shooting the moon") would be ill-advised. The general raison d'etre of this plan (the WHY facet) is to minimize the points P will take. This general plan recommends two different strategies. The first one is to unload cards with point values on opponents, and this recommendation is rated excellent. The second strategy is to do whatever is possible simply to avoid winning tricks with points in them. This recommendation is rated very good, somewhat less desirable than the other one.

The next example illustrates the recommended strategy of avoiding points.

Example 3.

```
concept   AVOID-POINTS
          type        HEURISTIC
          context     TRICKINPROGRESS
          arguments   ((P PLAYER))
          when        T
          why         (MINIMIZE (POINTS P))
          recommend   (((DRAWOUT-QUEEN P) EXCELLENT)
                       ((AVOID-POINTS-NOW P) GOOD)
                       ((AVOID-POINTS-LATER P) VERYGOOD)
                       ((DONTPLAY-QUEEN P) WORST))
```

This concept is a heuristic, relevant to a particular player P and a current trick in progress. It is always relevant when P wants to minimize his points. There are four different ways it recommends to accomplish avoiding points. We will look at two of these. The first suggests trying to draw out the queen of spades. The second suggests avoiding points now, as opposed to later.

Example 4.

```
concept   DRAWOUT-QUEEN
          type        HEURISTIC
          context     TRICKINPROGRESS
          arguments   ((P PLAYER))
          when        (AND (IS P LEAD) (NOT QUEENPLAYED)
                       (NOT (CARDINHAND 'QS P)) (SPADES-SAFE P))
          why         (AVOID-POINTS-LATER P)
          recommend   (((HIGHCARDS (CARDSBELOW 'QS (HAND P)))
                              VERYGOOD))
```

This concept suggests playing the highest ranked spades in the player's hand that are ranked lower than the queen. This suggestion is relevant when the player leads the trick, the queen has not already been played, the player himself doesn't hold the queen and the player has enough spades to insure that he himself will not eventually be forced to win a trick with the queen of spades in it.

Example 5.

```
concept   AVOID-POINTS-NOW
          type        HEURISTIC
          context     TRICKINPROGRESS
          arguments   ((P PLAYER))
          recommend   (((FOLLOW-LOW P) VERYGOOD)
                       ((LEAD-LOW P) VERYGOOD))
```

This concept describes another way to avoid taking points. It suggests either leading a low card or following in suit with a low card.

Example 6.

```
concept   FOLLOW-LOW
        type         HEURISTIC
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         (AND (NOT (IS P LEAD))
                          (OR (AND (BAD-DISTRIBUTION P SUITLED)
                                   (NOT (LASTINTRICK)))
                              (POINTSIN CURRENTTRICK)))
        why          (AVOID-POINTS-NOW P)
        recommend    (((HIGHCARDS (CARDSBELOW HIGHCARD
                                   (HAND P))) VERYGOOD)
                      ((FOLLOW-WITH-LOWEST P) GOOD))
```

This concept describes how a player can follow in suit with a low card in his hand. It is relevant when the player isn't leading the trick and there are points in the trick already or, because the player knows there's an uneven distribution of cards in that suit among the various players, there's some possibility points will eventually be added to the trick. This heuristic recommends two alternatives. It prefers to have the player follow suit with the highest cards he has which are below the current high card in the trick. It recommends less strongly simply playing the lowest card the player can.

The examples thus far have illustrated the declarative representation of action-oriented knowledge. The following examples illustrate different sorts of object representations.

Example 7.

```
concept  VOID
        type         PREDICATE
        context      ROUNDINPROGRESS
        arguments    ((P PLAYER) (S SUIT))
        definition   (ZEROP (NUMBEROFCARDS (CARDSINSUIT
                     (HAND P) S)))
```

This example illustrates the description of a predicate that tests whether a player P is void of cards in a suit S. The next example illustrates a declarative representation of the function that returns the set of cards which a player holds in a specified suit.

Example 8.

```
concept  CARDSINSUIT
        type         FUNCTION
        returntype   SETOFCARDS
        context      ROUNDINPROGRESS
        arguments    ((C SETOFCARDS) (S SUIT))
        definition   (COLLECT C WHEN (SUITF *)=S)
```

Additional examples of knowledge representation occur in Appendix I. That appendix also traces the action of many concepts during an actual decisionmaking cycle.

4.4 ACCEPTING ADVICE

IAs should have the capacity to accept a variety of kinds of advice. We distinguish four areas in which an expert user might wish to provide advice: domain knowledge, implementation knowledge, behavioral criticisms, and conjectured new knowledge. These are considered briefly in the following subsections.

4.4-1 Accepting Domain Advice

Even in an apparently "simple" domain like Hearts, the types of advice the user can provide are numerous. The following table illustrates the different types of knowledge he might wish to supply:

PLAY RULES:
- Rules of the game
- Rules of play
- Scoring rules

BASIC DESIGNS FOR PLANS:
- Shooting the moon
- Playing low

PLAY-LOW HEURISTICS:
- When to play low
- Why to play low
- How to play low

SHOOT-THE-MOON HEURISTICS:
- When to
- Why to
- How to: Get the lead (before hearts are played) and keep it forever
  - Plan 1: Pass all losing hearts
  - Plan 2: Try to win tricks with your losing hearts
- Fall back contingency plans
- Things not to do

COUNTER-PLAY HEURISTICS:
- Interpreting others' plans
  - Rule 1:
    If method M is part of a plan for achieving goal G, and player P is observed to use M (or some component), infer that P is striving for G.
  - Rule 1.1:
    The likelihood that P is actually striving for G may be estimated with Bayes' theorem to be the product of the a priori

probability of P seeking G times the likelihood of M given G divided by the sum of such products over all G' & M' such that G' can be achieved by M'.
Inferring others' hands or likely cards
Interfering with others' plans

CHOOSING AMONG ALTERNATIVES:
Estimating threats
Estimating opportunities
Finding moves that improve chances of more than one opportunity or that decrease chances of more than one threat
Preference for certainty or simplicity
Preference for keeping options open
Preference for moves that aren't easily interpreted by opponents
Rules for combining preferences and diverse reasons for the same action
Rules for pruning and guiding evaluation of possible plans

SCORING HEURISTICS:
When you're the current low-scorer, try to put someone out of the game
Try to give points to the lowest scorers
When you're not the current low-scorer, try to prevent the game from ending

Table 1. The diversity of types of domain advice in the Hearts task.

Our approach will de-emphasize the problem of understanding natural language. Instead, we will attempt to translate pre-parsed naturalistic statements into well-formed knowledge representations. Section 4.5 describes our approach to interpreting informality; Section 4.6 explains how we intend to operationalize advice once it's understood; and Appendix IX provides some details of well-formedness criteria for general IA knowledge bases.

For the purpose of illustrating advice taking, we introduce a very simple example from the Hearts domain. Suppose we are advised to "Avoid taking points." We will assume input to the system of the form:

```
(VP     (AGENT  ?)
        (VERB   AVOID)
        (MOOD   IMPERATIVE)
        (OBJECT (VP     (AGENT  ?)
                        (VERB   TAKE)
                        (OBJECT POINTS))))
```

(Such parsings could be provided by currently available general parsers.) Assume further that the concept TAKE has been defined as a specialization of the concept ACTION as shown in Fig. 3, taken from (Mostow, 1978). The definition of "player take card" is "player move card to player's pile." This definition is represented by concept C0014, which is a specialization of the concept MOVE.

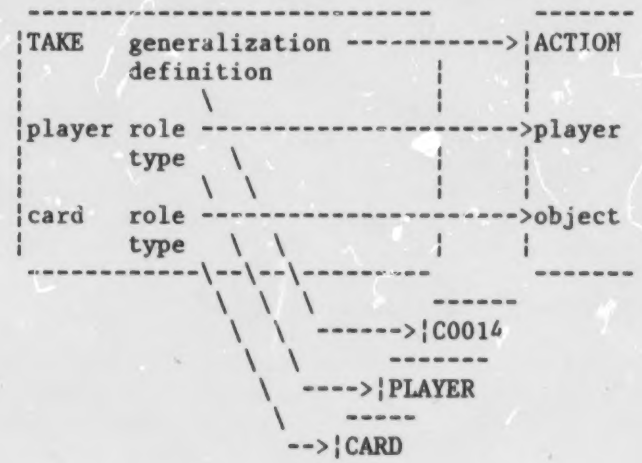

Figure 3: Representation of TAKE

By allowing consistently named concepts, facets, and types to inherit implicitly from their generalizations, we can simplify this notation to that shown in Fig. 4, taken from (Mostow, 1978).

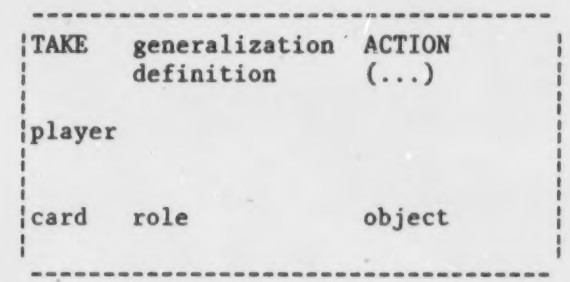

Figure 4. Role and type implicit in facet name.

To understand why a straightforward approach to representing "avoid taking points," e.g., in terms of comparable concept-pointer structures, would not suffice, we need to address the problem of interpreting informality. That is the topic of the Section 4.5. Before then however, we briefly discuss the other kinds of advice that arise.

4.4-2 Implementation Knowledge

Our approach to implementation knowledge is to develop a built-in stock of AI methods and programming techniques that can be mapped onto

closely matching domain knowledge descriptions. For example, we have developed a representation of the heuristic search method, and are now formulating descriptions of alternative control paradigms, such as "scripts." In this work we are trying to define criteria by which individual heuristics can be categorized as appropriate for application as generators, testers, demons, or as meta-heuristics to schedule the others. Then we anticipate being able to develop very general inference engines to evaluate (interpret) the heuristics. At present, we do not anticipate trying to take advice on implementation alternatives in a naturalistic format.

More details on implementation knowledge are provided elsewhere. Specifically, Sections 4.5 and 4.6 develop further the notion of operationalizing informally specified heuristics. Appendices III and IV discuss control schematizations and mechanisms for reordering computations to exploit demons and caching for the purposes of avoiding redundant computations.

4.4-3 Accepting Behavioral Criticisms

In our research on learning, we wish to allow the expert to point out incidents of poor behavior. Often, effective learning requires knowledge superior to that included in the IA itself. When the IA is performing some task, and has a complete context available to it, the expert's criticism can effectively delineate the problematic inferences and their causal antecedents. This would significantly improve the learner's capacity to focus on important problems. These problems are discussed in some detail in Appendices V and VI which provide an overview of the learning paradigm and a hand-simulated learning episode.

4.4-4 Accepting Conjectures

We anticipate that another important function an external expert can provide for us would be to conjecture new domain concepts. This would obviate the learner's task of generating those conjectures, but would still require the learner to implement and experimentally evaluate them.

4.5 INTERPRETING INFORMAL ADVICE [1]

Our basic approach to interpretation extends the functional semantics suggested by Montague as a basis for understanding natural language. In particular, Montague showed how individual words could be viewed as functionals (functions whose domains contain functions). The advantage of this view stems from the desirability of applying word meanings to other word meanings to modify and extend them. In this way, an adjective-noun phrase would itself be represented by the functional derived immediately by applying the adjective meaning (a functional) to the meaning of the noun. Hobbs & Rosenschein (1978) illustrate how these ideas can be implemented directly in LISP.

---

[1] This section is taken from (Mostow, 1978) with minor revisions.

In our research, we cannot assume (as these previous researchers did) that functional definitions are hand-crafted LISP functions. Instead, we need to accept concept definitions, translate these into functionals, and apply them to interpret advice. Our basic tasks involve translating parsed definitions into conceptual representations and combining concepts as required by the advice statements.

Concepts themselves are represented as entities in a higher-order logic. Combinations of concepts correspond to functional expressions in this logic. Expressions must satisfy certain well-formedness criteria, such as type agreement between the arguments of a function and the expressions to which it is applied. Informal conceptual constructs are treated as ill-formed expressions in this logic. Special interpretation heuristics will be used to correct ill-formed structures and to fill in missing components.

To illustrate these ideas, consider the interpretation of "taking points" in the phrase "avoid taking points." Fig. 5 shows the initial (incorrect) interpretation of "taking points" as the concept C1. This structure violates the well-formedness criterion of type agreement: the type of C1.card is CARD but its definition is C2, which is a MEASURE, not a CARD. Measures are functions; POINTS is a function whose argument is of type CARD and whose result is of type INTEGER. Formally, C2 is a expression constructed by applying POINTS to an (unidentified) object of type CARD.

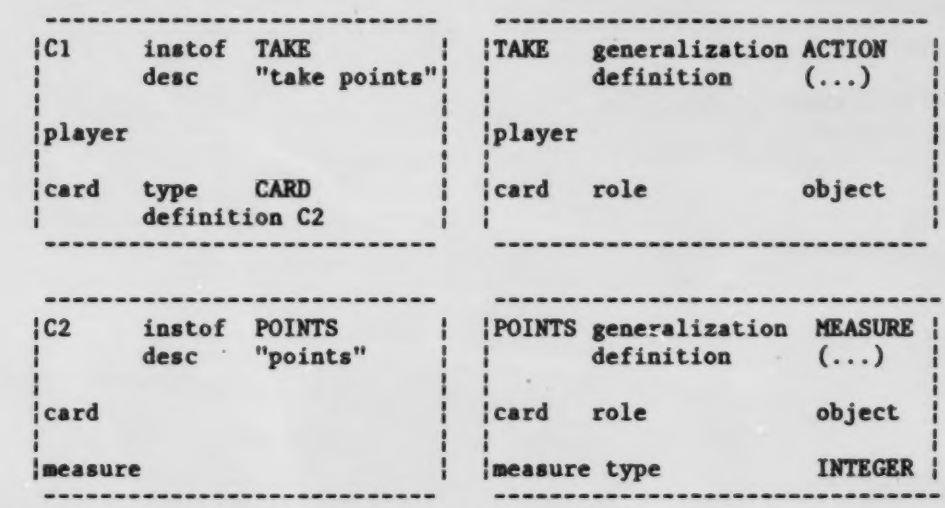

Figure 5: Initial (incorrect) interpretation of "take points".

In this situation, the known definition of "take" (originally supplied by the user as part of the definition of Hearts) applies to cards rather than to points. In the expression "taking points," the cards being taken are identified by one of their attributes, namely their points. Thus the correct interpretation can be found by a heuristic for interpreting attributive references:

> If a facet expects an object but receives a measure on it, try redefining it as an object whose measure is positive.

In the current example, C1.card expects a CARD but is defined as C2, an instance of POINTS; thus the heuristic suggests changing the definition of C1.card from C2 to C2.card, while restricting C2.measure to be a positive integer (POSINT). This revision is reflected in Fig. 6, which shows the interpretation of "avoid taking points" as the concept C0. Note that no definition is shown for the agent facet of C0. This reflects the parameterized nature of C0: the agent facet acts as a lambda variable, and can take on different values in different contexts.

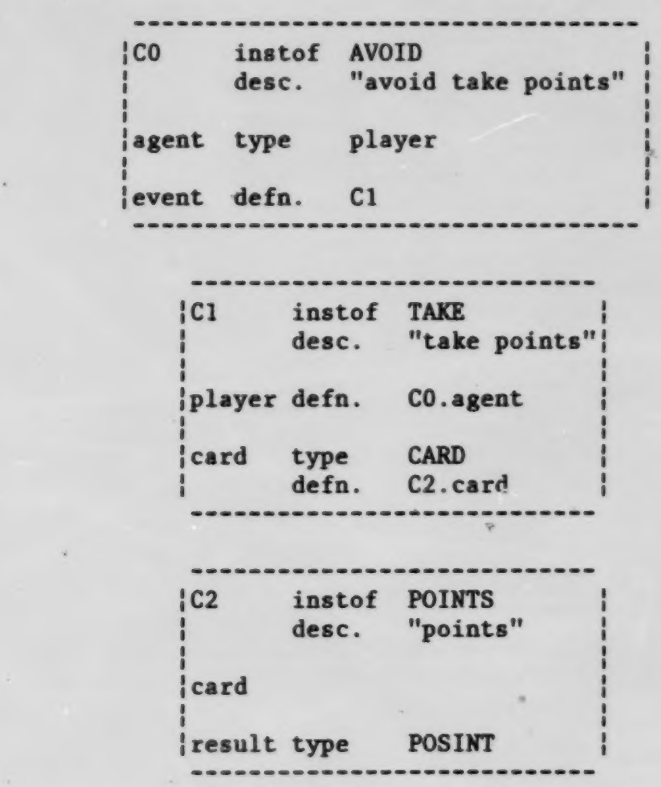

Figure 6: Interpretation of "avoid take points".

4.6 OPERATIONALIZING ADVICE [1]

One premise of our research is that AI provides a variety of suitable methods for operationalizing interpreted advice. Operationalization requires us to convert a definition of some concept to corresponding effective behavior. In essence, we must simulate or meta-evaluate our conceptual representations to obtain behavior that is consistent with them. As long as our objectives are limited IAs,

[1] For a more detailed discussion, see (Mostow, 1978).

we expect to be able to find effective and reasonably efficient methods for realizing their behaviors.

For example, the general heuristic search method can operationalize concepts of the form, "find a sequence satisfying some initial and final conditions." The problem statement for such methods specifies what information they require to operationalize a concept. For example, the heuristic search method requires a defined search space, an initial node in that space, some legal operators for transforming nodes in the space, and a test that the final node reached should satisfy. The rest of the method specifies how to apply the operators to the nodes to perform an effective search.

We will use a variety of such AI paradigms to operationalize concepts automatically. This requires an internal representation of the various methods. Thus far we have represented the heuristic search method and are currently working on a formalization of hypothesize-and-test. Our approach is to find one or more methods for each concept whose problem definitions closely match the components of the concept definition. Binding the corresponding components to the search method arguments should yield effective procedural operationalizations.

Problems that arise in this matching process include the following. First, the top-level concept definition may not directly match the problem statement. In this case, we may try to "view" the concept components in related, but somewhat abstracted ways. This type of problem occurs often in partial-matching tasks. Second, not all components of the problem statement may be explicit even in the correctly expanded concept definition. Again, a robust approach looks for a partial match and then uses inference rules to fill in missing correspondences, appealing to the user for assistance as needed.

4.7 CONTROL AND EFFICIENCY

We see our work as providing an excellent testbed for studying new ideas concerning control. Because it is so easy to specify what an IA should do and roughly how it should do it, most IA specifications will largely ignore sophisticated control issues. We believe this occurs because people do not typically think about general control methods for diverse collections of heuristic knowledge. Rather, when they approach problems requiring such varied types of knowledge, they behave more opportunistically than systematically.

Our primary interests in this area concern methods for using the same knowledge in various ways. We believe that most problem solving and domain knowledge can be separated from control principles. Thus, we wish to investigate ways of applying the same knowledge base in different ways. In particular, we have already explored applying heuristic knowledge in both top-down (exhaustive, systematic) ways and, alternatively, in a bottom-up-then-top-down (opportunistic, generate-and-test) control regime. Our declarative representations made this control variation easy. The differences in efficiency and psychological realism were remarkable (see Appendix III).

Two particular areas of research concern us. The first relates to the use of knowledge for inferences in different directions, particularly in top-down vs. bottom-up modes. The second relates to the transformation of knowledge-driven data flows to eliminate redundant inferences by saving state information. We refer to these problems as invertibility and caching, respectively. Each of these is discussed briefly below (see also Appendices III and IV).

4.7-1 Knowledge Invertibility

Several basic types of inference occur in IAs. In other domains, we have referred to these as recognition, prediction, and enumeration. Recognition requires finding that an existing data pattern instantiates a generic template or concept. Prediction often responds to the recognition of one concept by suggesting that related concepts will co-occur. Enumeration can translate a high-level general prediction into numerous alternative realizations of that prediction. In addition, some rating process must be used to evaluate hypotheses which represent uncertain inferences. Such functions are typical in speech and image understanding work, and we believe they will regulate IA activity in most other domains also.

All three types of inference are supportable or derivable from the same declarative knowledge. For example, in speech understanding, a grammatical phrase structure rule can be used to recognize valid word hypotheses, to predict adjacent phrase components or to re-express a predicted phrase in terms of its conjunctive or disjunctive constituents.

Similarly, the kinds of concept and heuristic representations we have developed can be applied in diverse ways. When the control regime is top-down (as in planning by top-down refinement), we can execute first the high-level plan heuristics, and successively execute those heuristics they recommend. Alternatively, we can execute those heuristics first that seem to match the current state and, later, attempt to find higher-level heuristics that support their desirability vis-a-vis high-level goals. The first type of control would be like problem reduction; the second is quite similar to hypothesize-and-test. Thus, depending on which regime we adopt, the same heuristic can be used either to generate hypotheses (recognize, predict, or enumerate) or to test hypotheses (evaluate them).

Another type of desirable invertibility can be derived from the declarative conceptual representation. This concerns the use of the same type of knowledge either to generate one's own behavior or to interpret that of others. For example, we can use a set of advised planning heuristics either to produce our own actions or to diagnose and interpret the apparent plans underlying an opponent's behavior. This type of inversion is advised in the sample heuristics in Section 4.4-1.

4.7-2 Caching and Focus of Attention

We refer to the action of storing previously computed results, in an effort to expedite their subsequent accessibility, as caching.

This type of function is apparent in most intelligent human behavior, but is difficult to obtain in typical AI systems. Current practices of saving previous results require a programmer to decide which results should be stored, where to store them, and when to change them.

We believe that a primary means of focusing attention effectively derives from the ability to limit responses chiefly to changing and surprising data. Thus, an ability to save previously computed results until they become invalid would be an essential ingredient in a change-driven control regime. Aside from its obvious efficiency aspects, the elimination of redundant inferences should also enable IAs to reach better results in limited time than they otherwise could. In this sense, focus achieves preferred results through the mechanism of limiting attention to the most critical and timely aspects of a situation.

Because our knowledge base definitions of IAs include both planning and problem solving heuristics, as well as conceptual representations of the domain entities and predicates, the KB provides sufficient information to diagnose opportunities for caching. In particular, whenever a frequently tested predicate evaluates some condition that changes rarely, we suppose the predicate would be a likely candidate for caching. The basic method of caching involves generating a demon concept to watch for changes in the evaluated conditions and to set a flag in the corresponding concept's cache. When a predicate or function is called, its cached value is returned directly unless the flag is on. In that case, the more complete definition is evaluated, the flag is turned off, and the demon is reactivated. These processes are defined in more detail in Appendix IV.

4.8 LEARNING: KNOWLEDGE ACQUISITION FROM INSTRUCTION AND EXPERIENCE

4.8-1 Learning as a Programming Problem

In this paper, we are embarking on a slightly different approach to learning than that pursued in our previous research. Previously we have viewed learning primarily as a generalization and statistical classification problem. In that work we have developed formal methods for abstraction, partial-matching, generalization and classification based on both feature and propositional descriptions. The results of such learning usually consisted of simple IF-THEN rules whose THEN-components were of one uniform type. In that sense, our learning mechanisms previously addressed homogeneous knowledge acquisition problems. Although the mechanisms we developed were applied also to a few complex rule induction problems, we expended little effort in that direction.

In this research, however, we focus on learning as a programming problem. The two primary sources of knowledge are seen to be advice and experience. In the case of advice, we say we have learned the expert's advice when we can behave in accordance with it. Aside from

understanding the advice, all that is required is to program ourselves to execute that advice. Once our initial behavior begins, however, we must recognize and correct its shortcomings. This corresponds to re-programming our knowledge. To accomplish this reprogramming requires abilities to criticize existing knowledge and to conjecture improved concepts.

4.8-2 Design of the Learner

The design of the learner encompasses the following activities: criticism, diagnosis, conjecture, validation and incorporation. We will discuss each process in turn. A learning scenario using these processes is given in Appendix VI.

Criticism:

The learner must be able to analyze and criticize the system's behavior. To accomplish this it must first be able to identify good and bad consequences of the actions taken. The learner should explicitly know the system's goals, plans, and expectations, and should be examining them throughout the activity (e.g., Hearts game). When certain goals are accomplished, the learner should recognize that the system's actions were successful and that the heuristics used were appropriate and should consequently be reinforced. When certain goals and expectations are not met, the learner must identify the consequences that did occur so that it may diagnose what happened.

Similarly, the learner should be able to incorporate an expert's criticism of the system's actions. It should be able to compare and contrast the system's behavior with the advice given by the expert, and incorporate his knowledge into its set of heuristics.

Thus, a system can improve its performance by criticizing and improving its bad play and by reinforcing its good play. But the learner should not stop there. It should be able to develop new, higher-level heuristics and plans that combine, generalize, specialize, and abstract existing heuristics and plans. Thus it should be able to examine successful actions and learn new, more sophisticated heuristics that embody the successful heuristics. Criticism does not just mean diagnosing mistakes, which is certainly an important and difficult task, but also means identifying aspects of good performance as well.

Diagnosis:

Once the learner has identified various consequences of its actions, it must diagnose what led to these consequences. The heuristics, plans, evaluations, etc. that led to these consequences must be known and available for examination. The system must be able to explain its processing--how and why alternative actions were chosen and used. A thorough and succinct representation of knowledge facilitates this objective, as the scenario in Appendix VI shows. Thus, for any situation, the learner must be able to work backward to locate the various decision points and reasons certain actions were taken.

One method for detecting and diagnosing consequences is through the use of demons. Heuristics and plans can specify what they expect to happen as a result of their use. These expectations could be in the form of conditions expected to occur as the game progresses. These conditions would be embedded in demons that watch the game and fire when the conditions are met or refuted. These demons would then point to the heuristics, plans, or evaluations whose expectations apparently require modification to improve the validity of the IA's knowledge.

Conjecture:

At this point, the learner has identified what heuristics were used that led to certain consequences. Further, it has also identified the alternative bypassed actions available at that time. Conjecture involves hypothesizing possible solutions either by modifying the heuristics at that decision point so that alternative actions would be taken (if bad consequences resulted) or by reinforcing the heuristics that led to good consequences or by formulating new higher-level heuristics to expand on the successful heuristics.

In order to modify existing heuristics and to formulate new heuristics, the learner must have substantial knowledge about learning methods. It must know how to constrain or relax conditions applicable to heuristics, how to generalize and to specialize heuristics, how to combine heuristics into higher-level heuristics, how to change evaluation measures, how to integrate recommendations, etc. A major effort will be involved in formulating and implementing a substantial set of such learning methods. Appendix VII provides a sample set of learning heuristics.

Validation:

Once heuristics are modified, the learner must validate that the modifications correct or improve the consequences that were originally criticized. If changes were made to correct bad consequences, the learner could simulate the same game beginning at the diagnosed decision point and confirm that the new or modified heuristics select actions which do not lead to the bad consequences first discovered. The new heuristics could, however, lead to other bad consequences. The learner must be able to recognize when this occurs. Otherwise it may find itself in a recursive loop, correcting heuristics that corrected other heuristics, etc.

In the case of the learner formulating improved heuristics to elaborate heuristics that resulted in good consequences, it must similarly validate that the same actions continue to occur, once again through simulation. But the learner also needs to monitor the new heuristics in subsequent games to see whether they have substantial impact and lead to favorable outcomes. If not, the new heuristics should be discarded.

The process of validation requires considerable effort and analysis. A human expert could significantly enhance this process. The expert could examine the new heuristics developed by the learner. His advice and criticism would be valuable in determining the appropriateness and applicability of the discovered heuristics. It may work both ways of course, because the learner should suggest generalized heuristics that he may not even have thought of.

Incorporation:

The final goal of the learner is to incorporate the new heuristics into the system. This is the function of the Knowledge Programmer. Learning, in this case, means that the system rather than the expert formulates the heuristics. The learner should monitor and criticize the new heuristics, as well as the pre-existing heuristics. Resources permitting, the learner should remain active. It should always be involved in analyzing and criticizing system behavior. Bad heuristics should be recognized and modified or discarded. Good heuristics should be reinforced, elaborated and generalized.

5. RELATED PREVIOUS RESEARCH

In this section, we briefly review related research results that support our current investigations. First we discuss our own and then the research of others.

5.1 OUR PREVIOUS RESEARCH

The current research effort extends four lines of research we have pursued for several years. Our primary areas of research related to the current project include: learning, planning, knowledge representation, and heuristic game playing. Each of these is discussed briefly in turn.

5.1-1 Learning

Our work on learning has focused on methods for abstracting generalities from multiple examples (Hayes-Roth, 1974, 1976a, 1976b, 1977, 1978a, 1978b; Hayes-Roth & Burge, 1976; Hayes-Roth & McDermott, 1976, 1978). This has led to descriptive languages with formal semantics and to related processes for induction, including interference matching and partial-matching methods. We have also developed numerous alternative data structures that have limited descriptive power but expedite these induction processes. Most of this work has aimed at the automatic formation of pattern recognition templates, although some of our research has explained how problem solving rules (e.g., rules of transformational grammar) could be inferred from conceptually annotated (parsed) exemplary sentence-pairs. In the larger context of the current project this type of inductive inference can be seen as a limited class of heuristic methods for acquiring new knowledge from repeated examples. Our work has influenced colleagues in both computer science and psychology, and the number of researchers in these topics continues to expand.

5.1-2 Planning

Our recent work on planning has developed a complex but comprehensive model of planning processes (B. Hayes-Roth & F. Hayes-Roth, 1978). This model emphasizes the use of diverse kinds of heuristics (e.g., strategies, tactics, problem solving models, policies) which interact through a blackboard structure like that in Hearsay-II (Erman & Lesser, 1975; Hayes-Roth, Fox, & Mostow, 1978). This planning model extends previous research in an important way. Specifically, most prior research on planning cast the problem into a uniform problem solving formalism (such as GPS or top-down refinement). As a result, many of the higher-level types of advice that planners exchange between themselves could not be incorporated directly into the planning programs. Since we anticipate an increased future emphasis on advising "how" as well as "what," we expect increased utilization of comparably rich planning formalisms. We have implemented two planners in INTERLISP in this framework, and are currently exploring its suitability as a framework for developing a handcrafted IA to assist in Air Force tactical planning.

5.1-3 Knowledge Representation

We have had a long-standing interest in knowledge representation. Our early learning research employed one of the first conceptual representations or "frame" languages to describe training examples and inferred concepts (Hayes-Roth, 1974). These descriptions were called PSRs (parameterized structural representations). The elements of PSRs were instantiated case frames, which correspond to concepts with facets and typed variables. PSRs have served us well in earlier learning research, but appear somewhat too limited for the more complex demands of the current work. Thus, we have already developed more elaborate and formally interpretable knowledge representations for use in this work. We have also been heavily involved in the development and application of production rules and other forms of pattern-directed inference systems for IAs (Hayes-Roth & Mostow, 1975, 1976a, 1976b; Hayes-Roth, Waterman & Lenat, 1978; Mostow & Hayes-Roth, 1978; Waterman & Hayes-Roth, 1978). Although we have helped pioneer their applications, we have also faulted rule-based representations for their inability to separate adequately domain knowledge from control principles (Mostow & Hayes-Roth, 1978). We anticipate that the proposed work will advance this field in important ways as we continue to improve our understanding of appropriate devices for separating control from knowledge per se.

In addition, we have developed two of the few working modularized focus of attention mechanisms, namely that in the Hearsay-II system (Hayes-Roth & Lesser, 1976, 1978) and in the PATHFINDER System (Klahr, 1978). These works provide useful principles for identifying aspects of knowledge that indicate its dynamic desirability for inclusion in inference processes. Such notions arise again in the proposed work on IAs, as we attempt to distinguish types of heuristics that would best play the roles of generators, testers, and evaluators.

On the semantics of conceptual representations, we have also conducted significant prior research. Two types of problems have received our attention. First, we have conducted research into the automatic compilation of knowledge from an extensible source such as the dictionary. Since that knowledge is informal and self-referential, we addressed initially the problems of combining word definitions to interpret longer phrases. This work was moderately successful for an arbitrary collection of words (McDonald & Hayes-Roth, 1978). What was important was our attempt to develop functional semantics, like those explored in our current work, without the use of natural language parsing or abstract semantic primitives. This was accomplished by using a semantic network representation of dictionary knowledge derived directly from the surface form of the concept definitions. The second area of research concerns a more elaborate and formal conceptual framework for knowledge representation. To support our current research, we have developed a more formal conceptual representation, like that illustrated in the previous section (Mostow, 1978) and are now defining its formal semantics.

5.1-4 Heuristic Game Playing

During the current year, we have tested our ideas about multifaceted heuristics for behavior in the domain of the card game Hearts. We have developed three systems. The first was a handcrafted IA incorporating about 15 heuristics that required 2 man-months to program and tune. The second was reimplemented using an initial conceptual representation of knowledge and a top-down control regime. This also required about 2 man-months. The third used a new control regime with the existing conceptual knowledge representation. This required less than one man-month. While the current IA includes fewer than 50 heuristics, we have avoided efforts to expand the set by handcrafting additional ones. At this point, we believe we understand what is necessary to automate the incorporation of additional heuristics, especially those of types not currently included. Our interest in game playing is minimal. We focus on the more generally useful task of acquiring knowledge in the hope that our results can favorably impact a wide community of AI researchers.

We have also analyzed numerous kinds of learning heuristics that people bring to bear to criticize behavior and conjecture new knowledge in the Hearts domain. We have collected protocols from both experts and novices to ascertain some of their methods. In particular we have found that experts exploit their superior knowledge of play, acquired primarily from instruction by other experts or slowly from extended experience, to formulate criticisms and improvements. Novices on the other hand use quite weak heuristics. (See Appendixes V and VI.)

5.2 RELATED RESEARCH OF OTHERS

The research effort we propose relates to a wide range of past and existing work. We have been influenced by, and will continue to share ideas with, many researchers and projects. Rather than scatter references throughout the paper, we have decided to gather them into a single section. The descriptions that follow will be brief and discussed primarily from the research perspective of our current project.

The research being performed at the Heuristic Programming Laboratory headed by Ed Feigenbaum at Stanford University shares with us the design goal of developing intelligent knowledge-based systems and agents. The work has involved the development of expert decision programs in various domains using a predominantly rule-based formalism, e.g, MYCIN (Shortliffe, 1976), DENDRAL (Feigenbaum, et al., 1971), and TIERESIAS (Davis, 1977, 1978). The work on PROSPECTOR (Duda, et al., 1978) at SRI is also in this mold. We wish to expand and enhance this research by exploring alternative representations of knowledge, the ability to acquire informally specified knowledge, the capability of learning new knowledge by the system's own analysis of its behavior, and the design of alternative methods of applying, controlling, and explaining knowledge.

The representation of knowledge as frames (see also Moore and Newell, 1974), e.g., KRL (Bobrow & Winograd, 1977) and FRL (Roberts & Goldstein, 1977), scripts (Schank & Abelson, 1977), concepts (Lenat, 1976), semantic nets (Woods, 1975), (Brachman, 1978), MOLGEN's units package (Martin, et al., 1977), and messages in SMALLTALK (Goldbert & Kay, 1972) and PLASMA (Hewitt, 1977), gives several alternative but similar methods for storing knowledge. We have been heavily influenced by Lenat's approach to representation in terms of the conceptualization of knowledge and will expand on his formalism and scope. In our initial exploration of conceptual representation, we have found it to be a very compact and extensible representation (see Appendix I for examples). We will extend the use of concepts to embody plans, strategies, evaluations, and interpretations of behavior. Representing interpretations of behavior is a new and exciting research area that needs expansion and development (cf. Carbonell, 1978; Waterman, 1978).

The work in automatic programming is following three primary directions. Green (1976) and Manna and Waldinger (1975) begin with formal specifications of desired programs and attempt to build programs to accomplish those specifications. The SAFE project (Balzer, et al., 1977) and the EP system (Waterman, 1978) take informal specifications of desired programs as initial input. In this respect they are similar to our desired method of acquiring knowledge. But the programs developed are algorithms rather than heuristics. Our work is more akin to the view of automatic programming as the formation of simulation specifications. Natural language input is mapped into problem specifications for a particular programming environment. The inference engine then drives the simulation process. In Simon's Understand project (Simon, 1977) natural language is mapped into GPS (Newell & Simon, 1972) formalism; in Heidorn's system (Heidorn, 1976), it is mapped into GPSS. Our view of acquiring, learning, and applying knowledge is also one of mapping into a programming environment--informal statements into concepts for the Knowledge Programmer and Learner.

Although research in self-learner systems began over 20 years ago with Samuel's checker program (1963), limited progress has been made. Only recently has momentum begun building. Vere (1978a; 1978b), Soloway and Riseman (1977), Mitchell (1977), and Winston (1975) have been concerned with learning general descriptions from specific examples. The learning heuristics employed in these projects were few and relatively specialized to limited knowledge bases. Only Lenat's learning heuristics (Lenat, 1976, 1977a, 1977b, 1978) reflect significant variability and generality.

Similarly, the area of knowledge acquisition has been recently getting more attention, after a long period of neglect following McCarthy's pioneering work on the advice taker (McCarthy, 1968). For knowledge-based systems to be intelligent, they must have expert knowledge. First, and foremost, this knowledge must come from the expert. Sophisticated learning mechanisms can only be used if some knowledge already exists. Recent work in knowledge acquisition (Davis, 1977; Zobrist, 1973; Michie, 1978) shows considerable promise, but the suggested techniques are in their infancy. Continued investigation along these lines is warranted.

The ability to diagnose behavior is crucial for learning mechanisms. Work on automatic debugging of programs (Sussman, 1975), planning and re-planning--e.g., STRIPS (Fikes et al., 1972), ABSTRIPS (Sacerdoti, 1974), NOAH (Sacerdoti, 1975), and finding faults in electronic circuits (Sussman, 1975, 1977), SOPHIE (Brown & Burton, 1975)--are initial explorations into automatic diagnosis. Hearsay-II diagnosed appropriate credit assignments to knowledge sources leading to correct and incorrect solutions in speech understanding. Our work is more focused on learning and incorporating new knowledge. In this respect, the learner must not only diagnose, but conjecture new heuristics. The learner is diagnosing heuristics which are rarely completely correct. It is not just a matter of finding a bug which is there or is not. Learning is itself a heuristic process, and one is seldom sure if the new heuristics are appropriate. Thus learning also involves validation through experience. Lakatos's brilliant work (Lakatos, 1976) focuses on the diagnostic process -- how knowledge is developed by analysis and experience. Even though his domain of interest was mathematics, his proposed techniques for converting proof analyses into refined concepts and deductions seem equally relevant in heuristic problem solving domains. We intend to formulate his heuristics for discovery in corresponding ways for our IA domains.

ACKNOWLEDGMENTS

Doug Lenat has provided valuable insights into several aspects of the research proposed. Notable areas among these include the conceptual representation of knowledge, the formulation and use of learning heuristics, and caching.

This work was supported in part by NSF Grant MC577-03273 to the Rand Corporation.

REFERENCES

Balzer, R., Goldman, N., and Wile, D. Informality in program specifications. Proc. 5th Int. Joint Conf. Artificial Intelligence, Cambridge, Mass., 1977, 389-397.

Bobrow, D. G. and Winograd, T. An overview of KRL, A knowledge representation language. J. Cognitive Sci., 1, 1977, 3-46.

Brachman, R. J. A structural paradigm for representing knowledge. Report 3605, Bolt, Beranek, and Newman, Cambridge, Mass., 1978.

Brown, J. S. and Burton, R. R. Multiple representations of knowledge for tutorial reasoning. In D.Bobrow and A. Collins (Eds.), Representation and Meaning. New York: Academic Press, 1975, 311-349.

Buchanan, B. G. and Mitchell, T. Model directed learning by production rules. In D. A.Waterman and F. Hayes-Roth (Eds.), Pattern-Directed Inference Systems. New York: Academic Press, 1978, 297-312.

Carbonell Jr., J. G. POLITICS: automated ideological reasoning. J. Cognitive Sci., 2, 1978, 27-51.

Davis, R. and King, J. An overview of production systems. In E. W. Elcock and D. Michie (Eds.), Machine intelligence 8. New York: Wiley, 1976, 300-332.

Davis R. Interactive transfer of expertise: Acquisition of new inference rules. Proc. 5th Int. Joint Conf. Artificial Intelligence, Cambridge, Mass., 1977, 321-328.

Davis, R. Knowledge acquisition in rule-based systems: Knowledge about representation as a basis for system construction and maintenance. In D. A.Waterman and F. Hayes-Roth (Eds.), Pattern-Directed Inference Systems. New York: Academic Press, 1978, 99-134.

Duda, R. O., Hart, P. E., Nilsson, N. J., and Sutherland, G. Semantic network representations in rule-based inference systems. In D. A.Waterman and F. Hayes-Roth (Eds.), Pattern-Directed Inference Systems. New York: Academic Press, 1978, 203-221.

Engelmore, R. S. and Nii, H. P. A knowledge-based system for the interpretation of protein x-ray crystallographic data. Heuristic Programming Project Report HPP-77-2 (also STAN-CS-77-589), Stanford Univ., Stanford, California, 1977.

Erman, L. D., and Lesser, V. R. A multi-level organization for problem solving using many, diverse, cooperating sources of knowledge. Proc. 4th Int. Joint Conf. Artificial Intelligence, Tbilisi, USSR, 1975, 483-490.

Feigenbaum, E. A., Buchanan, B. G., and Lederberg, J. On generality and problem solving: A case study using the DENDRAL program. In B. Meltzer and D. Michie (Eds.), Machine Intelligence 6. New York: American Elsevier, 1971, 165-190.

Fikes, R. E., Hart, P. E., and Nilsson, N. J. Learning and executing generalized robot plans. Artificial Intelligence, 3, 1972, 251-288.

Goldberg, A. and Kay, A. SMALLTALK-72 User's Manual. Xerox Palo Alto Research Center, Palo Alto, California, 1972.

Green, C.C. The Design of the PSI Program Synthesis System. Proc. 2nd Int. Conf. Software Engineering, San Francisco, 1976, 4-18.

Hayes-Roth, B. and Hayes-Roth, F. Cognitive processes in planning. The Rand Corporation, Santa Monica, 1978.

Hayes-Roth, F. Schematic classification problems and their solution. Pattern Recognition, 6, 1974, 105-114.

Hayes-Roth, F. Patterns of induction and associated knowledge acquisition algorithms. In C. H. Chen (Ed.), Pattern Recognition and Artificial Intelligence, Academic Press, New York, 1976a.

Hayes-Roth, F. Representation of structured events and efficient procedures for their recognition. Pattern Recognition, 8, 1976b, 141-150.

Hayes-Roth, F. Uniform representations of structured patterns and an algorithm for the induction of contingency-response rules. Informat. Control, 33, 1977, 87-116.

Hayes-Roth, F. The role of partial and best matches in knowledge systems. In D. A. Waterman & F. Hayes-Roth (Eds.), Pattern Directed Inference Systems. New York: Academic Press, 1978a.

Hayes-Roth, F. Learning by example. In A. M. Lesgold, et al. (Eds.), Cognitive psychology and instruction. New York: Plenum, 1978b.

Hayes-Roth, F. and Burge, J. Characterizing syllables as sequences of machine-generated labelled segments of connected speech: A study in symbolic pattern learning using a conjunctive feature learning and classification system. Proc. 3rd Int. Joint Conf. Pattern Recognition, Coronado, California, 1976, 431-435.

Hayes-Roth, F. and Lesser, V. R. Focus of attention in a distributed logic speech understanding system. Proc. 1976 IEEE Int. Conf. Acoustics, Speech and Signal Processing Philadelphia, Pennsylvania, 1976, 416-420.

Hayes-Roth, F., & Lesser, V. R. Focus of attention in the Hearsay-II system. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 1977.

Hayes-Roth, F. and McDermott, J. Learning structured patterns from examples. Proc. 3rd Int. Joint Conf. Pattern Recognition, Coronado, California, 1976, 419-423.

Hayes-Roth, F. and McDermott, J. An interference matching technique for inducing abstractions. Comm. ACM, 21, 6, 1978, 401-410.

Hayes-Roth, F. and Mostow, D. J. An automatically compilable recognition network for structured patterns. Proc. 4th Int. Joint Conf. Artificial Intelligence, Tbilisi, USSR, 1975, 246-251.

Hayes-Roth, F. and Mostow, D. J. Organization and control of syntactic, semantic, inferential and world knowledge for language understanding. Proc. 1976 Int. Conf. Computational Linguistics, Ottawa, Canada, 1976a.

Hayes-Roth, F. and Mostow, D. J. Syntax and semantics in a distributed logic speech understanding system. Proc. 1976 IEEE Int. Conf. Acoustics, Speech, and Signal Processing, Philadelphia, Pennsylvania, 1976b, 421-424.

Hayes-Roth, F., Mostow, D. J., and Fox, M. S. Understanding speech in the Hearsay-II system. In L. Bolc (Ed.), Natural Language Communication With Computers. Berlin: Springer-Verlag, 1978.

Hayes-Roth, F., Waterman, D. A., & Lenat, D. Principles of Pattern-Directed Inference. In D. A. Waterman & F. Hayes Roth, (eds.). Pattern-directed inference systems. New York: Academic Press, 1978.

Heidorn, G. E. Automatic programming through natural language dialog: a survey. IBM J. Research and Devlopment, 20, 1976, 302-313.

Hewitt, C. and Smith, B. Toward a programmer's apprentice. IEEE Trans. Software Eng., 1, 1975, 26-45.

Hewitt, C. Viewing control structures as patterns of passing messages. Artificial Intelligence, 8, 1977, 323-364.

Hobbs, J. R. and Rosenschein, S. J. Making computational sense of Montague's intensional logic. Artificial Intelligence, 9, 1977, 287-306.

Klahr, P. Planning techniques for rule selection in deductive question-answering. In D. A.Waterman and F. Hayes-Roth (Eds.), Pattern-Directed Inference Systems, New York: Academic Press, 1978, 223-239.

Lakatos, I. Proofs and refutations. Cambridge: Cambridge Univ. Press, 1976.

Lenat, D. BEINGs: Knowledge as interacting experts. Proc. 4th Int. Joint Conf. Artificial Intelligence, Tbilisi, USSR, 1975, 126-133.

Lenat, D. AM: An artificial intelligence approach to discovery in mathematics as heuristic search. SAIL AIM-286, Stanford Artificial Intelligence Laboratory, Stanford, California, 1976. Jointly issued as Computer Science Dept. Report No. STAN-CS-76-570.

Lenat, D. Automated theory formation in mathematics. Proc. 5th Int. Joint Conf. Artificial Intelligence, Cambridge, Mass., 1977, 833-842.

Lenat, D. The ubiquity of discovery: 1977 computers and thought lecture. Proc. 5th Int. Joint Conf. Artificial Intelligence, Cambridge, Mass., 1977, 1093-1105.

Lenat, D. and Harris, G. Designing a rule system that searches for scientific discoveries. In D. A.Waterman and F. Hayes-Roth (Eds.), Pattern-Directed Inference Systems. New York: Academic Press, 1978, 25-51.

McCarthy, J. The advice taker. Reprinted in M. Minsky (Ed.,), Semantic Information Processing, MIT Press, Cambridge, Mass., 1968, 403-410.

McDonald, D., & Hayes-Roth, F. A semantic network inferential approach to the design of an extensible language understanding system. In D. A. Waterman & F. Hayes-Roth (Eds.), Pattern-Directed Inference Systems. New York: Academic Press, 1978.

Manna, Z. and Waldinger, R. Knowledge and reasoning in program synthesis. Artificial Intelligence, 6, 1975, 175-208.

Martin, N., Friedland, P., King, J., and Stefik, M. Knowledge base management for experiment planning in molecular genetics. Proc. 5th Int. Joint Conf. Artificial Intelligence, Cambridge, Mass., 1977, 882-887.

Michie, D. and Bratdo, I. Advice table representations of chess end-game knowledge. Proc 3rd AISB/GI Conf on Artificial Intelligence, 1978, 194-200.

Mitchell, T. M. Version spaces: A candidate elimination approach to rule learning. Proc. 5th Int. Joint Conf. Artificial Intelligence, Cambridge, Mass., 1977, 305-310.

Moore, J., & Newell, A. How can Merlin understand? In L. W. Gregg (ed.), Knowledge and Cognition. New York: Lawrence Erlbaum Associates, 1974.

Mostow, J. Machine-aided heuristic programming: an AI approach to knowledge engineering. Department of Computer Science, dissertation proposal, Carnegie-Mellon University, 1978.

Mostow, D. J., & Hayes-Roth, F. A production system for speech understanding. In D. A. Waterman & F. Hayes-Roth (eds.), Pattern-Directed Inference Systems. New York: Academic Press, 1978.

Newell, A. and Simon, H. A. Human Problem Solving, Englewood Cliffs, N.J.: Prentice-Hall, 1972.

Nii, H. P., and Feigenbaum, E. A. Rule-based understanding of signals. In D. A.Waterman and F. Hayes-Roth (Eds.), Pattern-Directed Inference Systems, New York: Academic Press, 1978, 483-501.

Roberts, R. B. and Goldstein, I. P. The FRL primer. MIT Memo 408, Cambridge, Mass., 1977.

Sacerdoti, E. D. Planning in a hierarchy of abstraction spaces. Artificial Intelligence, 5, 1974, 115-135.

Sacerdoti, E. D. The nonlinear nature of plans. Proc. 4th Int. Joint Conf. Artificial Intelligence, Tbilisi, USSR, 1975, 206-214.

Samuel, A. L. Some studies of machine learning using the game of checkers. In Feigenbaum, E. A. and Feldman, J. (Eds.), Computers and Thought. New York: McGraw-Hill, 1963, 71-105.

Schank, R. C., and Abelson, R. P. Scripts, Plans, Goals, and Understanding. Hillsdale, N.J.: Lawrence Erlbaum Associates, 1977.

Shortliffe, E. H. Computer-Based Medical Consultations: MYCIN. American Elsevier, New York, 1976.

Simon, H. A. Artificial intelligence systems that understand. Proc. 5th Int. Joint Conf. Artificial Intelligence, Cambridge, Mass., 1977, 1059-1073.

Soloway, E. M., and Riseman, E. M. Knowledge-directed learning. Proc. Workshop Pattern-Directed Inference Systems, SIGART Newsletter 63, 1977, 49-55.

Sussman, G. J. A Computational Model of Skill Acquisition. New York: American Elsevier, 1975.

Sussman, G. J., and Stallman, R. Heuristic techniques in computer aided circuit analysis, Memo 328, MIT AI Lab., Cambridge, Mass., 1975.

Sussman, G. J. Electrical design: A problem for artificial intelligence research. Proc. 5th Int. Joint Conf. Artificial Intelligence, Cambridge, Mass., 1977, 894-900.

Vere, S. A. Inductive learning of relational productions. In D. A. Waterman & F. Hayes-Roth (Eds.), Pattern-Directed Inference Systems. New York: Academic Press, 1978, 281-295.

Vere, S. A. Multilevel counterfactuals for generalizations of relational concepts and productions. Tech. Report, Univ. Illinois Chicago Circle, 1978.

Waterman, D. A. Generalization learning techniques for automating the learning of heuristics. Artificial Intelligence, 1, 1970, 121-170.

Waterman, D. A. Rule-directed interactive transaction agents: An approach to knowledge acquisition. R-2171-ARPA, The Rand Corporation, Santa Monica, California, 1977.

Waterman, D. A. Exemplary programming in RITA. In D. A. Waterman & F. Hayes-Roth (Eds.), Pattern-Directed Inference Systems. New York: Academic Press, 1978, 261-279.

Waterman, D. A. and Hayes-Roth, F. (Eds.). Pattern-Directed Inference Systems. New York: Academic Press, 1978.

Winston, P. H. Learning structural descriptions from examples. In P. H. Winston (Ed.), The Psychology of Computer Vision. New York: McGraw-Hill, 1975.

Woods, W. A. What's in a link: foundations for semantic networks. In D. G. Bobrow and A. Collins (Eds.), Representation and Understanding. New York: Academic Press, 1975.

Zobrist, A. I. and Carlson Jr., F. R. An advice-taking chess computer. Scientific American, 228, 1973.

Appendix I: CONCEPTUAL REPRESENTATION OF HEURISTICS

The following is a complete specification of the heuristics of the PLAY-LOW-PLAN (i.e., play-low -- don't take any points) currently implemented in a prototype Hearts player. The purpose here is to show how heuristics can be compactly and succinctly represented in the conceptual framework presented in this paper. The control methods used to apply the knowledge shown here are discussed in Appendix III. We first show a sample trace of the use of these heuristics in an actual game.

The following is taken from an actual game trace that shows what heuristics were used to make card selections. HP-2 is a machine player:

```
HP-2 LEADS TRICK 3
HP-2's hand:   (5C 6C 2D 3D 5D 7D 9D 10D JD QD KD 4H 8H KH 9S)

GENERATORS:
        (9S 13 DRAWOUT-QUEEN)
        (KD 10 LEAD-SAFE-WINNERS)
        (6C  5 TRY-TO-VOID-LEAD)

RECOMMENDATIONS:
        9S
          13:PLAY-LOW-PLAN
            14:AVOID-POINTS
              14:AVOID-POINTS-LATER
                10:TRY-TO-VOID
                13:DRAWOUT-QUEEN
              13:DRAWOUT-QUEEN

        KD
          11:PLAY-LOW-PLAN
            11:AVOID-POINTS
              11:AVOID-POINTS-LATER
                11:GETRID-WINNERS
                  10:LEAD-SAFE-WINNERS

        6C
          7:PLAY-LOW-PLAN
            7:AVOID-POINTS
              7:AVOID-POINTS-LATER
                5:TRY-TO-VOID-LEAD
                5:GETRID-WINNERS
                  5:LEAD-SAFE-WINNERS

HP-2 PLAYS...  9S
```

The trace first shows HP-2's hand. Generators are heuristics that initially hypothesize potentially good cards to play. They are discussed in Appendix III. Note how they reduce the number of cards actually considered (HP-2 can play any of its non-heart cards in this situation). Those cards hypothesized by the generators are then given

full consideration by all the heuristics. The integers shown in the generators and recommendations are numeric evaluation measures used to indicate the strength of the recommendations of the specified heuristics.

The indented format of the recommendations reflects the hierarchical structure of the heuristics. For example, in the recommendation of the 9S, the PLAY-LOW-PLAN suggested trying to AVOID-POINTS, which, in turn, recommended trying to AVOID-POINTS-LATER and to DRAWOUT-QUEEN. AVOID-POINTS-LATER suggested TRY-TO-VOID-LEAD and DRAWOUT-QUEEN. TRY-TO-VOID-LEAD recommended the 9S with strength 10; DRAWOUT-QUEEN recommended it with strength 13. AVOID-POINTS-LATER combined these recommendations to give a strength of 14 to the 9S. This process of combining recommendations continued up the heuristic tree until the top PLAY-LOW-PLAN gave its final recommendation for the 9S. The card with the highest recommendation is selected.

The heuristics in the PLAY-LOW-PLAN are:

```
concept   PLAY-LOW-PLAN
        type         PLAN
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         (OR (TAKE OPPONENT POINTS)
                         (LACKING P HIGHCARDS)
                         (NOT (CONTROL P HEARTS)))
        why          (MINIMIZE (POINTS P))
        recommend    (((UNLOAD-POINTS P) EXCELLENT)
                      ((AVOID-POINTS P) VERYGOOD))


concept   UNLOAD-POINTS
        type         HEURISTIC
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         (OR (IS P LEAD) (VOID P SUITLED)
                         (IS SUITLED 'H) (IS SUITLED 'S))
        why          (GETRID P POINTS)
        recommend    (((DUMP-QUEEN P) BEST)
                      ((DUMP-HEARTS P) VERYGOOD))


concept   DUMP-QUEEN
        type         HEURISTIC
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         (CARDINHAND 'QS P)
        why          (UNLOAD-POINTS P)
        recommend    (((SAFE-LEAD-QUEEN P) BEST)
                      ((DISCARD-QUEEN P) BEST)
                      ((SAFE-QUEEN-FOLLOW P) BEST))
```

```
concept   SAFE-LEAD-QUEEN
        type         HEURISTIC
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         (AND (IS P LEAD) (CARDINHAND 'QS P)
                          (ONLY-AK-SPADES-OUT P))
        why          (UNLOAD-POINTS P)
        recommend    (('QS BEST))


concept   DISCARD-QUEEN
        type         HEURISTIC
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         (AND (VOID P SUITLED) (CARDINHAND 'QS P))
        why          (UNLOAD-POINTS P)
        recommend    (('QS BEST))


concept   SAFE-QUEEN-FOLLOW
        type         HEURISTIC
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         (AND (IS SUITLED 'S) (CARDINHAND 'QS P)
                          (OR (CARDINSET 'AS CURRENTTRICK)
                              (CARDINSET 'KS CURRENTTRICK)))
        why          (UNLOAD-POINTS P)
        recommend    (('QS BEST))


concept   DUMP-HEARTS
        type         HEURISTIC
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         (OR (IS P LEAD) (VOID P SUITLED)
                         (IS SUITLED 'H))
        why          (UNLOAD-POINTS P)
        recommend    (((DISCARD-HEARTS P) EXCELLENT)
                      ((LEAD-SAFE-HEARTS P) VERYGOOD)
                      ((FOLLOW-LOW P) EXCELLENT))


concept   DISCARD-HEARTS
        type         HEURISTIC
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         (VOID P SUITLED)
        why          (UNLOAD-POINTS P)
        recommend    (((HIGHCARDS (CARDSINSUIT 'H (HAND P)))
                             EXCELLENT))


concept   LEAD-SAFE-HEARTS
        type         HEURISTIC
        context      TRICKINPROGRESS
```

```
        arguments   ((P PLAYER))
        when        (AND (IS P LEAD) (LEGAL-HEART-PLAY))
        why         ((GETRID P POINTS) (GIVEUP P LEAD))
        recommend   (((SAFE-CARDS (CARDSINSUIT 'H (HAND P)) P)
                      VERYGOOD))


concept   AVOID-POINTS
        type        HEURISTIC
        context     TRICKINPROGRESS
        arguments   ((P PLAYER))
        when        T
        why         (MINIMIZE (POINTS P))
        recommend   (((DRAWOUT-QUEEN P) EXCELLENT)
                     ((AVOID-POINTS-LATER P) VERYGOOD)
                     ((AVOID-POINTS-NOW P) GOOD)
                     ((DONTPLAY-QUEEN P) WORST))


concept   DONTPLAY-QUEEN
        type        HEURISTIC
        context     TRICKINPROGRESS
        arguments   ((P PLAYER))
        when        (AND (CARDINHAND 'QS P)
                         (OR (IS SUITLED 'S)
                             (AND (IS P LEAD)
                             (NOT (ONLY-AK-SPADES-OUT P))))
                         (NOT (CARDINSET 'AS CURRENTTRICK))
                         (NOT (CARDINSET 'KS CURRENTTRICK)))
        why         (AVOID-POINTS P)
        recommend   (('QS WORST))


concept _ DRAWOUT-QUEEN
        type        HEURISTIC
        context     TRICKINPROGRESS
        arguments   ((P PLAYER))
        when        (AND (IS P LEAD) (NOT QUEENPLAYED)
                     (NOT (CARDINHAND 'QS P)) (SPADES-SAFE P))
        why         (AVOID-POINTS-LATER P)
        recommend   (((HIGHCARDS (CARDSBELOW 'QS (HAND P)) T)
                      VERYGOOD))


concept   AVOID-POINTS-LATER
        type        HEURISTIC
        context     TRICKINPROGRESS
        arguments   ((P PLAYER))
        when        T
        why         T
        recommend   (((GETRID-WINNERS P) VERYGOOD)
                     ((DRAWOUT-QUEEN P) EXCELLENT)
                     ((TRY-TO-VOID-LEAD P) EXCELLENT)
                     ((TRY-TO-VOID P) EXCELLENT)
                     ((DONT-PLAY-SPADES P) VERYBAD))
```

```
concept   GETRID-WINNERS
        type         HEURISTIC
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         T
        why          (AVOID-POINTS-LATER P)
        recommend    (((LEAD-SAFE-WINNERS P) VERYGOOD)
                      ((DISCARD-WINNERS P) EXCELLENT)
                      ((FOLLOW-HIGH P) VERYGOOD)
                      ((FORCED-HIGH-FOLLOW P) GOOD)
                      ((FOLLOW-HIGH-BELOW-QUEEN P) GOOD)
                      ((DONTPLAY-AK-SPADES P) VERYBAD))


concept   LEAD-SAFE-WINNERS
        type         HEURISTIC
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         (IS P LEAD)
        why          (GETRID-WINNERS P)
        recommend    (((HIGHCARDS (CARDSINSUIT 'C (HAND P)))
                              (GOOD-DISTRIBUTION P 'C))
                      ((HIGHCARDS (CARDSINSUIT 'D (HAND P)))
                              (GOOD-DISTRIBUTION P 'D))
                      ((LEAD-SAFE-SPADES P)
                              (GOOD-DISTRIBUTION P 'S)))


concept   LEAD-SAFE-SPADES
        type         HEURISTIC
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         (AND QUEENPLAYED (IS P LEAD))
        why          (GETRID-WINNERS P)
        recommend    (((HIGHCARDS (CARDSINSUIT 'S (HAND P)))
                              (GOOD-DISTRIBUTION P 'S)))


concept   DISCARD-WINNERS
        type         HEURISTIC
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         (VOID P SUITLED)
        why          (GETRID-WINNERS P)
        recommend    (((DISCARD-AK-SPADES P) EXCELLENT)
                      ((HIGHCARDS (HAND P) T) GOOD))


concept   DISCARD-AK-SPADES
        type         HEURISTIC
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         (AND (VOID P SUITLED)
                          (NOT (CARDINHAND 'QS P))
                          (NOT QUEENPLAYED)
```

```
                    (CARDSABOVE 'QS (HAND P)))
        why         (GETRID-WINNERS P)
        recommend   (((CARDSABOVE 'QS (HAND P)) BEST))


concept   FOLLOW-HIGH
        type        HEURISTIC
        context     TRICKINPROGRESS
        arguments   ((P PLAYER))
        when        (AND (NOT (IS P LEAD))
                         (NOT (POINTSIN CURRENTTRICK))
                         (OR (LASTINTRICK)
                             (AND (GOOD-DISTRIBUTION P SUITLED)
                                  (OR (NOT (IS SUITLED 'S))
                                      QUEENPLAYED))))
        why         (GETRID-WINNERS P)
        recommend   (((HIGHCARDS (CARDSINSUIT SUITLED (HAND P)))
                      EXCELLENT))


concept   DONTPLAY-AK-SPADES
        type        HEURISTIC
        context     TRICKINPROGRESS
        arguments   ((P PLAYER))
        when        (AND (OR (IS SUITLED 'S) (IS P LEAD))
                         (OR (CARDINSET 'QS CURRENTTRICK)
                             (AND (NOT QUEENPLAYED)
                                  (NOT (LASTINTRICK)))))
        why         (AVOID-POINTS P)
        recommend   (((CARDSABOVE 'QS (HAND P)) VERYBAD))


concept   FORCED-HIGH-FOLLOW
        type        HEURISTIC
        context     TRICKINPROGRESS
        arguments   ((P PLAYER))
        when        (AND (LASTINTRICK)
                         (NOT (CARDSBELOW HIGHCARD (HAND P))))
        why         (GETRID-WINNERS P)
        recommend   (((HIGHCARDS (CARDSINSUIT SUITLED (HAND P)))
                      GOOD))


concept   FOLLOW-HIGH-BELOW-QUEEN
        type        HEURISTIC
        context     TRICKINPROGRESS
        arguments   ((P PLAYER))
        when        (AND (IS SUITLED 'S) (NOT QUEENPLAYED))
        why         (GETRID-WINNERS P)
        recommend   (((HIGHCARDS (CARDSBELOW 'QS (HAND P)))
                      GOOD))
```

```
concept   TRY-TO-VOID
        type         HEURISTIC
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         (VOID P SUITLED)
        why          (AVOID-POINTS-LATER P)
        recommend    (((PLAY-SINGLETON-SUITS P) EXCELLENT)
                      ((HIGHCARDS (PLAY-DOUBLETON-SUITS P)) GOOD))


concept   TRY-TO-VOID-LEAD
        type         HEURISTIC
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         (IS P LEAD)
        why          (AVOID-POINTS-LATER P)
        recommend    (((PLAY-SINGLETON-SUITS P 'C)
                             (GOOD-DISTRIBUTION P 'C))
                      ((PLAY-SINGLETON-SUITS P 'D)
                             (GOOD-DISTRIBUTION P 'D))
                      ((PLAY-SINGLETON-SUITS P 'S)
                             (GOOD-DISTRIBUTION P 'S))
                      ((HIGHCARDS (PLAY-DOUBLETON-SUITS P 'C))
                             (GOOD-DISTRIBUTION P 'C))
                      ((HIGHCARDS (PLAY-DOUBLETON-SUITS P 'D))
                             (GOOD-DISTRIBUTION P 'D)))


concept   AVOID-POINTS-NOW
        type         HEURISTIC
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         T
        why          T
        recommend    (((FOLLOW-LOW P) VERYGOOD)
                      ((LEAD-LOW P) VERYGOOD))


concept   FOLLOW-LOW
        type         HEURISTIC
        context      TRICKINPROGRESS
        arguments    ((P PLAYER))
        when         (AND (NOT (IS P LEAD))
                          (OR (AND (BAD-DISTRIBUTION P SUITLED)
                                   (NOT (LASTINTRICK)))
                              (POINTSIN CURRENTTRICK)))
        why          (AVOID-POINTS-NOW P)
        recommend    (((HIGHCARDS (CARDSBELOW HIGHCARD (HAND P)))
                             VERYGOOD)
                      ((FOLLOW-WITH-LOWEST P) GOOD))


concept   FOLLOW-WITH-LOWEST
        type         HEURISTIC
        context      TRICKINPROGRESS
```

```
        arguments   ((P PLAYER))
        when        (AND (NOT (IS P LEAD)) (NOT (LASTINTRICK))
                         (NOT (CARDSBELOW HIGHCARD (HAND P))))
        why         (FOLLOW-LOW P)
        recommend   (((LOWCARDS (CARDSABOVE HIGHCARD (HAND P)))
                          GOOD))


concept  LEAD-LOW
        type        HEURISTIC
        context     TRICKINPROGRESS
        arguments   ((P PLAYER))
        when        (AND (IS P LEAD) (BAD-DISTRIBUTION P))
        why         (AVOID-POINTS-NOW P)
        recommend   (((SAFE-CARDS (HAND P) P) VERYGOOD)
                     ((DONT-PLAY-SPADES P) VERYBAD))


concept  DONT-PLAY-SPADES
        type        HEURISTIC
        context     TRICKINPROGRESS
        arguments   ((P PLAYER))
        when        (AND (IS P LEAD) (NOT QUEENPLAYED)
                         (OR (CARDINHAND 'QS P)
                             (NOT (SPADES-SAFE P))))
        why         (AVOID-POINTS-NOW P)
        recommend   (((CARDSINSUIT 'S (HAND P)) VERYBAD))
```

A brief description of the functions, predicates, and variables used in the heuristics:

Variables:

- BEST, EXCELLENT, VERYGOOD, GOOD, BAD, VERYBAD, WORST
  numeric values ranging from 30 to -30 used as recommendation strengths
- CURRENTTRICK - list of cards in the current trick
- SUITLED - suit of the card led in the current trick
- LEAD - player leading in the current trick
- HIGHCARD - highest card in the suit led in the current trick
- HEARTSBROKEN - whether hearts have been broken yet
- QUEENPLAYED - whether the queen of spades has been played yet

Functions and predicates:

- IS - are its two arguments the same
- AND - the conjunction of its arguments
- OR - the disjunction of its arguments
- NOT - the negation of its argument
- VOID - is the specified player void in the specified suit
- HAND - the hand of the specified player
- CARDINHAND - is the specified card in the specified player's hand
- CARDINSET - is the specified card in the specified set of cards
- POINTSIN - number of points in a given set of cards

LASTINTRICK - is current player the last player in current trick

CARDSBELOW - cards below a specified card from a given set of cards

CARDSABOVE - cards above a specified card from a given set of cards

CARDSINSUIT - cards of a particular suit from a given set of cards

PLAY-SINGLETON-SUITS - set of cards in a player's hand that are in singeltons suits

PLAY-DOUBLETON-SUITS - set of cards in a player's hand that are in doubleton suits

HIGHCARDS - set of given cards ordered by rank

ONLY-AK-SPADES-OUT - determines if only AS and KS remain out in the spade suit relative to a given player

LEGAL-HEART-PLAY - determines if a heart play is currently legal

SAFE-CARDS - orders a set of cards by how safe they are (number of cards out below each card)

GOOD-DISTRIBUTION - the strength of good distribution of a player in a particular suit

BAD-DISTRIBUTION - the strength of bad distribution of a player in a particular suit SPADES-SAFE - how safe a player is in not taking the Queen of Spades if spades are led

Appendix II: THE DESIGN OF THE KNOWLEDGE PROGRAMMER [1]

This Appendix describes information flow within the Knowledge Programmer of Figure 7.

Advice from the Expert is first PARSED to extract its syntactic structure. This parsed input is then recorded in an internal representation designed to facilitate inference and INTERPRETED in the sense of (1) having its informalities and ambiguities removed and (2) being integrated with the other system knowledge units. The internal representation is then OPERATIONALIZED by finding one or more general methods (such as Heuristic Search) adequate to implement the advice. Once the knowledge has been operationalized, it is APPLIED to produce actual task behavior. This whole process interactively consults previously acquired knowledge and the user at any time; the CONSULT component mediates the latter.

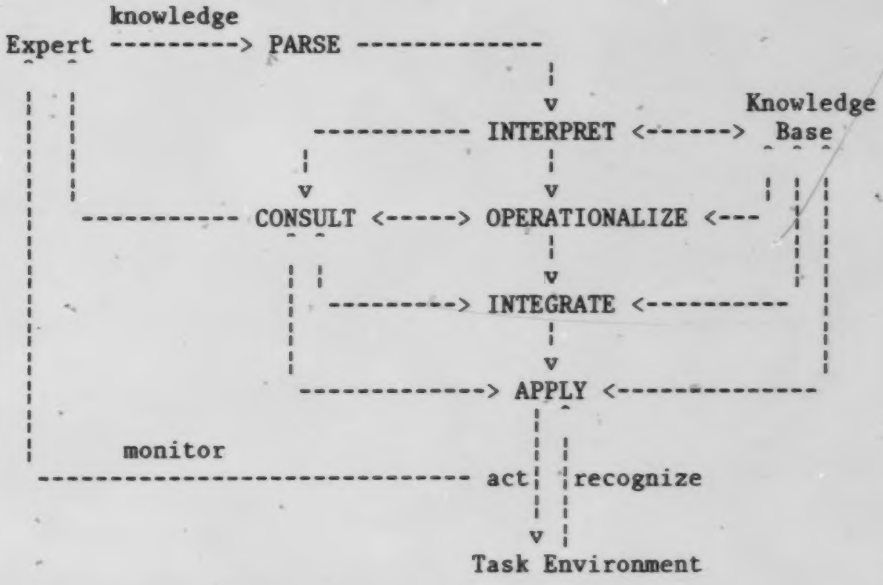

Figure 7: Information flow in Knowledge Programmer

Parsing:

The effort involved in providing a free-form English front-end for the Experts' interaction with the user would be considerable. Moreover, this would be immaterial to the main thrust of our work. We have adopted the solution of providing for only syntactically unambiguous pre-parsed input.

---

[1] This appendix is condensed from (Mostow, 1978) with minor revisions.

Knowledge as Concepts:

A concept (see Section 4.3) can be thought of as a generalized lambda expression. The facets themselves, representing terms in the expression, are sub-divided into aspects; each aspect describes features of the facet, such as its type, how to derive the value and so on. The entities filling the aspects are called "items." Our "aspects" are comparable with the "slotnames" of other schema representations. However, our concept-facet-aspect-item structure introduces an extra level in addition to the usual schema-slotname-slotfiller representation. We are thus able to capture the notion of clusters of slots containing closely related information, such as the type, value, and definition of some quantity. The "definition" is like the body of a lambda expression; it tells how to evaluate, apply, or recognize a concept. It may be another concept or a facet of some concept.

In addition to this structure within concepts, there is an external structure imposed by a conceptual hierarchy. Except for primitive concepts, every concept is derived from a more general concept by a derivation relation, such as generalization, specialization, individuation, or function application. The relationship between a concept and the one from which it is derived is strong. The derived concept must have facets corresponding to those of the general concept, and the mapping between corresponding facets must be indicated by the "role" aspect of the derived facets.

Interpretation:

Interpretation involves making sense of the informally-expressed utterances of the Expert. Our approach is a "functional semantics" approach. Concepts are represented as entities in a higher-order logic. Expressions must satisfy well-formedness criteria such as type agreement between the value returned by a function and the expression of which it is part. Informal utterances may now be treated as giving rise to ill-formed expressions in this higher-order logic. The task of intepretation is to correct the ill-formed structures with special heuristcs.

Operationalization:

During operationalization, the definition of a concept is used to derive effective behavior. Various general methods have been provided by AI for this. As an example, Heuristic Search can be used in operationalizing concepts such as "find a sequence of actions satisfying some constraint." The problem statement for a method specifies what information is necessary in order to operationalize a given concept using that method. In case of Heuristic Search, this requires the definition of a search space, the initial node in the space, the set of legal operators used in moving between nodes in the space, and the constraint to be satisfied by the resulting sequence. The method itself describes how to use these elements effectively in finding the solution.

We envisage using a combination of AI methods, flexibly choosing the one most appropriate for each situation. This flexibility is facilitated by the use of a declarative internal representation. The methods will be represented as procedurals, which are analogous to functionals. A functional takes some arguments and returns a function; a procedural takes some arguments and returns a procedure. Each parameterized component of a method is represented as an argument of a procedural.

The arguments of these procedurals may have types which seem at first to be unusually complicated -- for instance, "set of search pruning heuristics." But because of this, they can serve as sites for the incorporation of domain-specific knowledge, thus instantiating it for the problem at hand.

Integration:

Our system will need to accept a wide variety of advice, and be able to integrate it irrespective of its diversity or inconsistency. For example, Appendix VIII includes such heuristics as:

H1. If you're leading and going low and have enough spades to last without getting the queen of spades, then lead a spade below the queen.
H2. Any card that helps you void is a good card.
H3. Any card that can cause you to get the queen of spades is a bad card.
H4. Any card that is a guaranteed loser is a good card.
H5. Getting rid of guaranteed losers is undesirable unless necessary.
H6. Leading a low heart is good.

Our solution to the problems of the interaction between these pieces of (sometimes conflicting) advice is to separate as much as possible the implementation of different heuristics. Interactions between them are restricted to simple interfaces.

In order to do this, we adopt a uniform model of heuristics, so that each evaluates how well some action or object meets some criterion. The measure of how well it meets it is called the recommendation. For example, the heuristic "avoid taking points" would produce a recommendation of 100% for "playing the three of clubs." The model can also express a recommendation for such heuristics as "A player who leads a heart card early in the game is shooting for the moon." Additionally, it could even apply to heuristics by rating their reliability, utility, or computational efficiency.

Application:

The application of knowledge that has been interpreted, operationalized and integrated also requires careful consideration. While the initial version of the system may evaluate all heuristics in coming to a decision about which card to play, subsequent versions may

schedule only certain heuristics to be evaluated. An Agenda would be kept of heuristics to evaluate, ordered by the overall desirability of evaluating them (given their cost and utility) in the context of the immediate problem. An agenda organization allows the processes involved in the refinement of card choice to continue until the allotted computational power has been expended and it is time to continue with the external game.

Appendix III: CONTROL METHODS

Control refers to the mechanism that applies and manipulates knowledge. In the game of hearts, the control process would be responsible for examining the current state of the game, and the set of heuristics and plans in the knowledge base, to select intelligent moves.

One of our primary concerns is to achieve independence between knowledge and control. The knowledge base should be sufficiently descriptive and transparent to allow the use of alternative problem solving methods. This would allow an expert to specify particular problem solving strategies in various circumstances. Further, it would allow the system itself to select different control methods, examine their behavior, and learn which control methods are best suited for particular applications. For example, a generate-and-test paradigm might be suitable to select an initial game plan, while a backward-chaining paradigm might be suitable to implement the plan relative to the current hand.

In our second prototype Hearts program, we have designed and implemented two control strategies which we describe below. Our experience has shown that it is not only feasible to separate knowledge and control but that such separation allows a powerful modularization in the construction of knowledge-based systems. We were able to develop the second control strategy without any change to the knowledge base at all.

The initial control method used was a top-down control structure. Given a game state and a particular player to move, control would first extract the player's current game plan. The particular plan would suggest the use of certain heuristics to accomplish the goals of the plan (e.g., to play low, avoid taking points and dump points when possible). These heuristics would, in turn, recommend the use of other heuristics. This process would recurse until bottom-level heuristics were encountered. These heuristics would recommend particular cards or sets of cards (e.g., when void in the suitled, dump high cards or point cards). Recommendations would then be combined and a card would be selected.

The top-down control structure worked well but it had some obvious deficiencies. Extensibility was difficult. In order to add a new heuristic, it was necessary to find an appropriate place for it in the heuristic hierarchy. This involved fine tuning on the part of the programmer. Furthermore, since the control structure was top-down, the system was exhaustive in that it considered all heuristic paths and all potential cards as possible moves. This process seemed highly inefficient. It seemed that heuristics could immediately suggest that certain cards were potentially good possibilities, while others were definitely bad. The control structure should be more data-driven instead of data-blind as the top-down method appeared to be.

In order to overcome these deficiencies, a second control structure was designed and implemented. This control structure was based on the generate-and-test paradigm. Certain heuristics were

designated as "generators" and were used to scan the legal cards to determine if any would make good or bad candidates. The generators would produce hypotheses which represented initial recommendations. Those hypotheses recommending cards favorably remained in consideration while cards recommended negatively were discarded. Thus, the generators pruned the set of legal card plays to those with high potential value. The remaining heuristics served to support or refute the existing hypotheses. Recommendations were again evaluated and combined to produce a card play.

The use of generators to hypothesize cards greatly improved efficiency. Only a subset of the potential set of legal card plays was considered. The control process was no longer exhaustive and was more data-driven. It was also interesting to uncover another research area -- which heuristics should be considered generators and which heuristics should be testers. Since we felt that the control structure should be data-driven, we decided that those heuristics that recommended particular cards or sets of cards, rather than those recommending other heuristics, should be generators. The generators were then ordered by potential impact, i.e., the strength of their recommendations. Those cards recommended most strongly would thus be in the initial set of hypotheses.

Once a card was recommended in an hypothesis, it was decided that all the heuristics could serve as testers. Various generators could all support a particular hypothesis. But a particular card was hypothesized only once, by the generator with the highest recommendation for that card. In order not to exclude the recommendations of the other generators, they would be considered testers as well. Further research is needed to determine the criteria for distinguishing generators and testers in general.

It should be noted that neither the heuristics themselves, nor the way their recommendations were composed in choosing a card, were changed in any way in connection with changing the control method. Hence we have been able to demonstrate some independence between knowledge and control.

Both control methods were implemented procedurally, i.e., coded in LISP. Our goal is to make these control methods declarative. We have argued that the knowledge base should be declarative (with procedural attachments for function definitions). Knowledge represented this way is transparent and can be easily modified by the expert or by the system's learning process. We argue that this should be the case for control methods as well. We feel that the conceptual representation used to represent knowledge is rich enough to represent control and we will seek to conceptualize alternative problem-solving and control strategies.

	Control Method	
	Top-down	Generate-and-Test
Cards Evaluated	All	Best
Data-Blind?	Yes	No
Efficient?	No	Yes
Procedual?	Yes	Yes

Appendix IV: CACHING AND DEMONS

Applying a knowledge unit (KU) from the knowledge base (KB) involves the evaluation of functions and predicates. Often the values they return do not change between one application of the KU and the next. Such redundant evaluations are inefficient. The solution we propose is to cache, or store, these intermediate results. Caching becomes significant for those functions that are recomputed often within the same context and thus produce the same value. When a function is called, instead of executing or evaluating the function, its cached value is retrieved and returned.

Cached values can become outdated. A particular function depends on specific conditions in the environment. When these conditions change, the value of the function can also change. Thus, cached values are only valid when the certain conditions the function depends on remain the same. Once these conditions change, the function should be recomputed when called (instead of retrieving existing cached values). The recomputed value can now be similarly cached until it is outdated. Two questions emerge:

1. When is caching appropriate?
2. What is the mechanism for detecting whether cached values are still valid or are outdated?

Given a set of arguments, a function evaluates its definition relative to the arguments and returns a value, i.e., for a particular function F,

F: arguments => value

Caching involves storing values of F corresponding to given arguments, i.e.,

F:
<arguments, value>
.
.
.
<arguments, value>

Thus, given a set of arguments, the associated value (of F) can be retrieved through a matching process that operates over the arguments.

Each function has a "definition," whether it be an expression to evaluate (e.g., a LISP lambda expression), a procedure to invoke, or compiled code to execute. Functions typically examine various aspects of the environment, i.e., the current "state," and return different values depending on the current state. When the state changes, a function's value, for the same set of arguments, may also change. However, this is not always the case. A function usually examines few aspects of the environment, namely those relevant to the function's definition. These relevant aspects can be represented as a set of predicates, or "conditions." When these conditions change, a

function's value relative to a set of arguments is no longer valid and must be recomputed through the function's definition. Thus, a cached value is dependent on the conditions and would look more like a triple:

<conditions, arguments, value>

or, if considering the global context, rather than certain conditions within that context:

<state, arguments, value>

In the Hearts task, for example, the "state" of the game changes each time a card is played. However most "conditions" change only occasionally or rarely. For example, some conditions concern the spade suit. Cards played in the other suits would not effect these conditions. Thus it would seem that cached values should be relative to conditions rather than states. This is true for cached values as the game progresses dynamically.

However, two important applications of cached values may need to store states rather than conditions. The first involves look-ahead, where immediate decisions examine future scenarios to see potential outcomes of various alternative actions. Hypothetical states of the game need to be generated and retrieved as the system looks ahead to evaluate future states and outcomes. The use of states may also be significant in learning, where the system needs to go back to earlier states of the game and examine aspects or conditions of those states that may have led to favorable or unfavorable consequences. We will explore the use of states in look-ahead and learning.

We have designed a mechanism to monitor various conditions affecting cached values. The mechanism involves the use of demons that look for changes that invalidate cached values. Demons can effect cached values in two different ways:

1. Certain demons change cached values immediately, i.e., functions or variables are immediately reevaluated and the resulting values cached.

2. Certain demons record that various cached values are out of date, but those values are not recomputed unless they are needed (called) by the system.

The first type of demon is the most obvious way of changing cached values and can simply be looked at as a procedure which calls certain functions to update their values. An example would be a QUEEN-DEMON which follows the play of the game and notices when the queen of spades is played. As soon as it does, certain variables and conditions are immediately updated to reflect the consequences of the queen play, e.g., that hearts are broken and that spades are no longer a suit meriting special attention.

The second type of demon is the most significant since it delays reevaluation until the functions are actually called. Let's consider a cached value to be a triple as described above:

<conditions, arguments, value>

A demon would activate when relevant conditions change in the environment. These changes would be recorded in the conditions to indicate that function values are out of date. Now when a function is called with a set of arguments, the cached value (triple) associated with the arguments is retrieved. The conditions are checked, and if out of date, the function's definition is reevaluated relative to the arguments and re-cached with conditions also updated. Note that this process changes the cached value for the given arguments. The other cached values are not altered or examined. Cached values needed at any particular moment are the only ones accessed and possibly changed.

An example of this second type of demon would be SPADE-DEMON. This demon notices when spades are played during the game. It effects the function SPADES-SAFE which is used to indicate how safe a player is in avoiding taking the queen of spades when spades are led. If a player has a spade higher than the queen, he is in danger of potentially taking the queen. In this case, SPADES-SAFE is calculated according to the number of spades he has relative to the average number of spades per player, i.e., whether he has enough spades to play before he has to play his high ones where he would endanger taking the queen. The distribution of spades changes when spades are played. Thus SPADE-DEMON records that SPADES-SAFE is out of date when spades are played. The next time SPADES-SAFE is accessed for that player, it is reevaluated to produce an updated value.

Demons are either active or passive. For example, both QUEEN-DEMON and SPADE-DEMON are initially active. When the queen of spades is played, QUEEN-DEMON fires. Aside from changing appropriate variables and functions, QUEEN-DEMON deactivates itself since the queen of spades will not be played again in the current round and it deactivates SPADE-DEMON since SPADES-SAFE is no longer relevant. Demons are indexed by their triggering conditions. These conditions are examined during play and if satisfied, they trigger the appropriate demons, e.g., a spade play triggers SPADE-DEMON.

Caching is also a suitable subject for learning. The system could monitor its own function activations and note the frequency of changed values. Those functions returning the same values with respect to given arguments are good candidates for caching. The system could then look at a function's definition to locate conditions examined by the function and set these up as conditions for a demon activation.

Caching, with its demons to notice when cached values are outdated, is also integral to the design of the Learner, which uses demons to encode expectations in the same way. In fact, caching can be seen as a general method of organizing data flow, rather than simply a way of obtaining efficiency. Consider each heuristic as a production. Then all of the condition sides could be merged into a

discrimination net, with some subset of paths being active at each cycle. Organizing demons to update the truth-status of the tests would be a natural way of implementing such a system. Even if conditions are not "shared" as in a discrimination net, ready reference to cached values seems to be a sensible KU invocation paradigm. Moreover, there are cognitive parallels. Human attention seems sensitive to changes in the environment, so that noticing some change can lead to a restructuring of the approach to the situation. Furthermore, not noticing the ramifications of a change is a frequent cause of failure in problem solving, so it seems that people do effectively "cache" values. Indeed, this is one of the major uses made of human memory--to store partial results--as a form of cognitive economy.

Appendix V: DESIGN FOR A LEARNING WORKBENCH

Learning mechanisms may be classified as active or passive. A passive mechanism can only record observations from an environment. Although its processes may include such powerful techniques as abstraction, generalization and so on, it cannot act upon the environment and so its learning process is wholly responsive to external events. An active learning mechanism, on the other hand, can act upon the environment to refine its Knowledge Base. Hence an active learning mechanism has control over its learning experiences and their order. It also controls the methods (and variations on methods) it uses internally to generate its behavior and so it can obtain detailed feedback upon specific Knowledge Units (KUs) in the KB. In short, active mechanisms allow a learner to experimentally evaluate its hypotheses.

Other things being equal, a more active process is likely to learn faster than a more passive process. There is evidence that animals and people (especially children) use an active approach. Because learning is inherently difficult, the approach taken here is to optimize the conditions for an active learning mechanism to improve a Knowledge Base by applying it to a Task Environment.

Hearts will provide examples where they are needed, as elsewhere in this paper (Appendix IX addresses the generality of such KBs).

First, we note that simply playing games and recording their outcomes will probably be too impoverished a source of data for refining a hearts KB. Numerous KUs will have been applied in generating each specific card play, and others still will be consistent with any card played. Moreover it will be the game which is won or lost, rather than any smaller unit of analysis; for instance, taking a heart may prevent a moon shoot. Hence the problem of credit assignment over whole games would admit only weak statistical techniques such as the "reinforcement" of KUs which seemed to aid successful outcomes.

Although they may be extracted from actual play, we are led to seek smaller units of analysis which pose specific problems for the Learner. For instance, the machine hearts player may suddenly win the queen of spades, where "suddenly" means that there was no demon set up with that expectation. Play could be interrupted and control handed to the learning mechanism. (Play may resume later in case some other expectations are violated; but this is by no means necessary.) Alternatively, we can envisage sprouting a co-routine to pursue the corresponding learning problem in parallel with subsequent play.

The game up to that point would then be analyzed in depth. If appropriate new knowledge units could be conjectured, they would be verified by re-running the game. In the case of human players, this would require fresh deals, but a program can obviate such requirements. If no conjecture was induced, the game could be re-run making different choices to avoid the unexpected outcome.

Once a solution was found, the game could again be replayed to the problem point with small variations on the internal structures. The conjectured and apparently validated improvement could have been due to chance.

Experimental game situations for these phases will be obtained by passing constraints to the Task Environment Simulator which will set up a new context (processing state) that satisfies them. For instance, the KB may not be sufficiently rich to answer such questions as "What if player X had three hearts, rather than two?" It is desirable to pose that question while the relevant context is in hand. The process of specifying and exploring experimental game situations is active in a different way from the activity inherent in choosing a card and predicting the consequences.

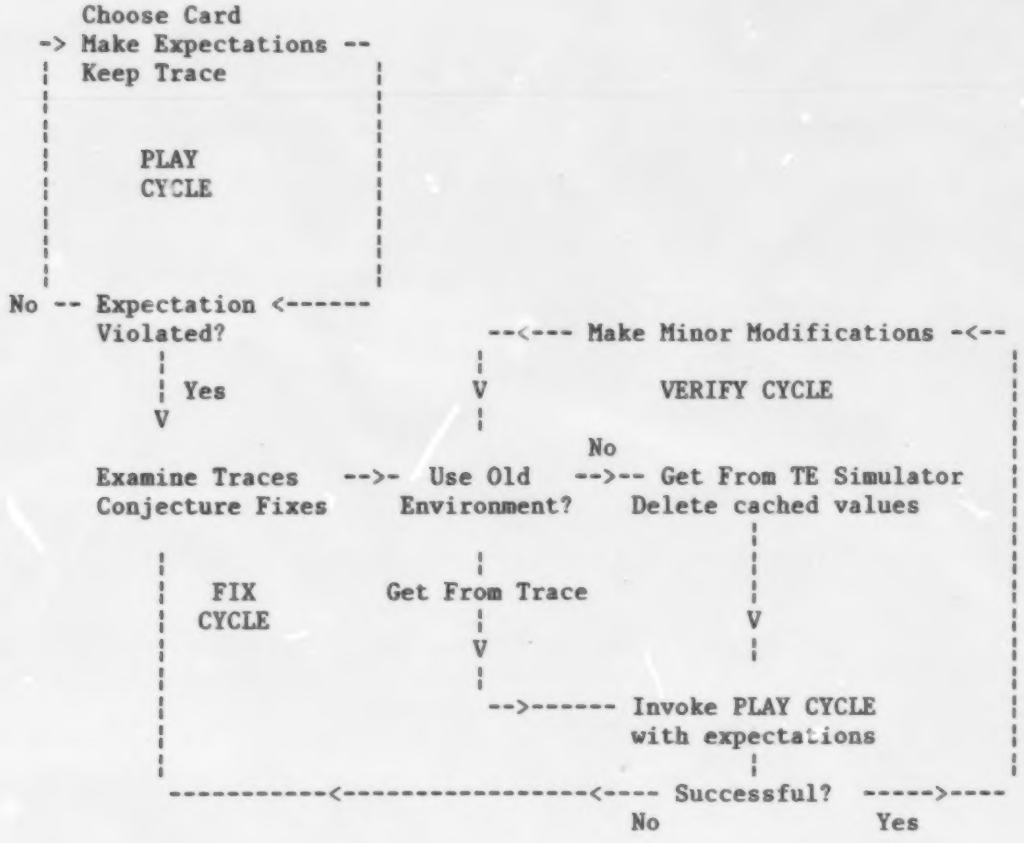

Figure 8: Control Flow in Learning Workbench

Exploring one game problem in depth in this way can lead to considerable efficiencies. Besides focus of attention (which is a learning efficiency), it can also be computationally efficient. Firstly, much of the information required by the KB may be cached (as

explained in Appendix IV), but more importantly, many of the bindings both from game situations to the KB and also within the KB may also be saved between replays.

Most importantly, though, it will lead to a precision in the characterization of knowledge which is unobtainable in any other way.

Appendix VI: A LEARNING SCENARIO

The following example scenario illustrates the learning process outlined in the paper. The example involves the heuristic DISCARD-WINNERS which recommends the play of WINNING-CARDs when a player is void in the suitled (i.e., get rid of your winners). A WINNING-CARD is a card that is likely to take a trick that contains points. At the outset WINNING-CARD is defined loosely as any "high card." The higher the rank of the card, the more likely the card is a winning card. We show how the Learner can modify the definition and recommendations of the DISCARD-WINNERS heuristic when it notices that the current definition is inadequate.

Player P's current hand includes AC, 3C, QD, and JD. A spade is led. P is void in spades and is considering what card to sluff. DISCARD-WINNERS recommends the AC since it is P's highest card. DISCARD-WINNERS generates a demon to watch what happens to the QD since this was DISCARD-WINNERS alternative selection. P selects AC to discard. Later in the game, a diamond is led, P plays the QD, a heart is discarded, and P takes the trick.

The Learner criticizes P's play. The Learner has a set of heuristics for criticizing plays. One of them states that taking points when playing low (his game plan) is bad. In this situation, P has taken points which the Learner criticizes.

The Learner must diagnose what went wrong. Once again, the Learner has a set of heuristics for diagnosis. One of the diagnostic heuristics suggests that alternative plays should be examined. P selected the QD when the diamond was led. The Learner recognizes that P had another option when diamonds were led, namely the JD. It then simulates the trick by having P play the JD and notices that the same result obtains: P still takes the trick and the point.

The Learner is still in the diagnosis stage. Another diagnostic heuristic suggests examining demons activated in the current trick. The demon generated by DISCARD-WINNERS has fired since the QD was played. The Learner diagnoses that DISCARD-WINNERS may be faulty. How can the Learner determine if this is indeed the case? Consider Figure 9.

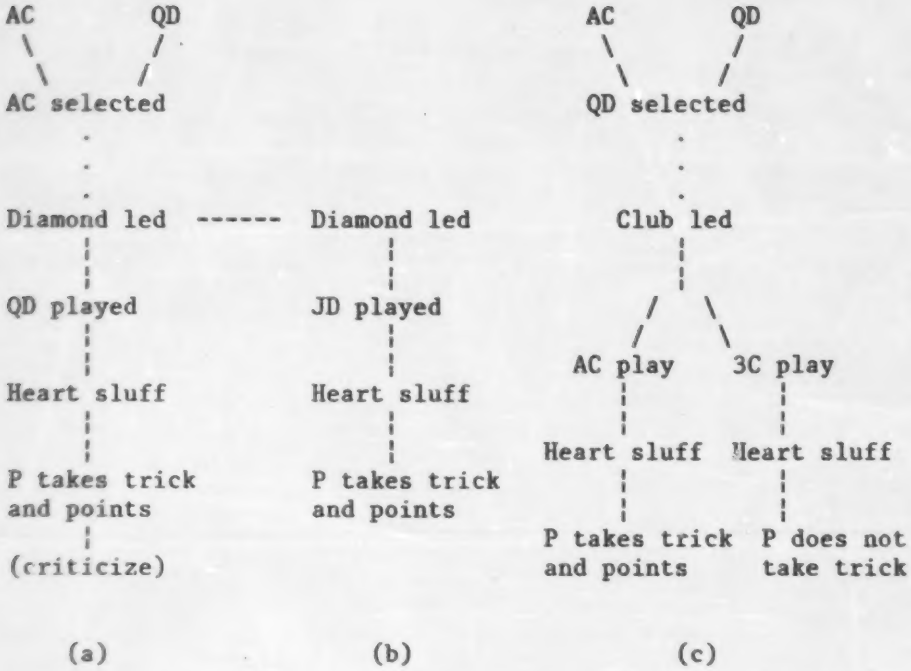

Figure 9. Criticism and Diagnosis

Figure 9a shows the actual play and the point at which the Learner criticized P's play. Figure 9b represents the Learner's first attempt at diagnosing the situation--considering P's alternative choice for the current trick. Figure 9c depicts the Learner's diagnosis of the DISCARD-WINNERS heuristic. The demon pointed to the DISCARD-WINNERS heuristic as the one that decided to keep the QD while discarding the AC. The Learner considers what would have happened if DISCARD-WINNERS selected its alternative choice, the QD, instead of the AC. The diagnostic heuristic used here suggests trying to show if an alternative choice (directed by the demon) eliminated the problem in an analogous game situation. The Learner then simulates the play of the game to be analogous to the actual play. This is shown in Figure 9c. (Note that another diagnostic heuristic could have suggested simulating the new play, QD instead of AC, and then continuing with the actual play of a diamond lead. This diagnostic branch is not considered here.)

If the QD were discarded earlier and a club is subsequently led (analogous to the actual play of the AC being played and a diamond led), P has two options, the AC and the 3C. The play of the AC once again results in P taking the trick and points. But the 3C play results favorably for P. He does not take any points. The Learner has found a scenario that eliminates the bad consequences originally encountered. It suggests that DISCARD-WINNERS should be modified. The Learner now has to conjecture possible solutions.

DISCARD-WINNERS recommends the play of WINNING-CARDs. The Learner recognizes that QD is a winning card. Thus it decides that DISCARD-WINNERS should continue to recommend WINNING-CARDs but other considerations seem to be important as well. The Learner has access to all the concepts (functions, heuristics, evaluations, variables, etc.) in the system's knowledge base. It must decide what it is about the 3C that made it possible to avoid taking points while the JD (as an alternative to the QD) led to the same negative consequences. It searches the concepts to find those that apply to the cards and suits being considered.

One of the conjecture heuristics suggests modifying recommendations to include conditions (concepts) that differentiate between actual recommendations and desired recommendations. Among the applicable concepts, the Learner discovers that 3C is a LOSING-CARD, that the club suit is a SAFE-SUIT since it contains a card (3C) that is a LOSING-CARD, that JD is a WINNING-CARD, and that the diamond suit is a DANGEROUS-SUIT since it contains only WINNING-CARDs. It wants DISCARD-WINNERS to select the QD instead of the AC as it currently does. The Learner now conjectures a possible solution -- DISCARD-WINNERS should recommend WINNING-CARDs in DANGEROUS-SUITs. The QD would be recommended since it is a winning card in a dangerous suit. The AC would not be recommended since it is a winning card in a safe suit.

Let's review the Learner's processing to this point:

1. Criticise -- the player took a trick containing points; this is bad since the player is playing low and trying to avoid taking points.

2. Diagnosis -- alternative options at the current trick failed to alleviate the problem; DISCARD-WINNERS decided to keep a particular card that eventually resulted in the player taking points; how could this card have been discarded earlier and what would have been the consequences of keeping the card that was originally discarded; simulating the possibilities showed that a scenario did exist that solved the problem.

3. Conjecture -- existing concepts were applied to the cards and suits to determine if any could distinguish between the various alternatives; concepts that would produce the selected play desired were added to the DISCARD-WINNERS heuristic.

The Learner must now validate that the modified heuristic selects the QD instead of the AC at the original decision point. The game state at that point is retrieved and the new DISCARD-WINNERS is used. The Learner validates that the desired card is selected by the new heuristic. It then must monitor DISCARD-WINNERS in subsequent plays to see how successful the modified version is in selecting good plays.

Once the Learner is confident that the modified heuristic performs the desired actions, it gives the heuristic to the Knowledge Programmer which implements the heuristic (incorporates the heuristic into the knowledge base).

Appendix VII: LEARNING HEURISTICS

Learning may be viewed as an active process of problem-solving, as a search through a space of concepts which is guided by feedback from their use. The following list of heuristics helps to optimize that process. Since our real interest is the general learning problem, it is couched in general terms, but since our exemplary domain is Hearts, the examples given are for Hearts. A truly general learning mechanism would be able to apply the general heuristics to the semantic descriptions of diverse tasks.

Initialization:

The system's task during the initialization phase is to make some first hypotheses about useful heuristics. LH1 takes the domain, here Hearts, and explores the implications of the semantics of the domain, here the rules of Hearts. However, not all possible heuristics derived in this way will be equally useful for learning. Heuristics must be widely-applicable (LH2), so that feedback will be obtained readily during the initial games. Also they must be easily evaluated (LH3), or the uncertainty involved in the evaluation will be confounded with the reliablity of the heuristic as a whole.

LH1. Generate initial heuristics by exploring the implications of the semantics of the domain.
LH2. Prefer widely-applicable concepts.
LH3. Prefer easily-evaluated concepts.

Examples of heuristics generated in this way might be:

HH1: High cards are dangerous
HH2: Get rid of dangerous cards at the first opportunity

Progress:

There must be some way of guaranteeing that the Learner will not stagnate. For instance, it could become satisfied with simply not losing points, and never take upon itself a more difficult task as a basis for refining its knowledge. LH4 encodes this consideration. However, it is also possible to stagnate while attempting a task that is difficult (a phenomenon similar to finding a local maximum in hill-climbing). LH5 attempts to counteract this by trying out new combinations of what is already well established, even though these new combinations have not been suggested by examples thus far provided by the environment. LH6 tries to ensure that KU's which contain a germ of the truth are not discarded before the opportunity arises to refine them in some concrete situation, and LH7 even proposes trying them sometimes as they stand--but only when the game is not at risk.

LH4. Try to do progressively more difficult things, e.g. at first, don't lose points; later, shoot the moon.
LH5. If no new concepts occur for some time, generate new from old and use them.
LH6. Keep unreliable concepts, but use only modifications of them.

LH7. Try using unreliable concepts, when it is safe to do so.

Feedback:

Feedback about the effectiveness of heuristics is a crucial but difficult problem for learning mechanisms. Providing for good feedback is a major factor in the design of the Learning Workbench (Appendix V) and is one motivation for the use of demons to encode expectations. The feedback heuristics make explicit our hypotheses on the solution to these problems.

It is important to extract as much as possible from each experience. LH8 says that the system should try to bring as much knowledge as possible to bear upon each decision. Some KUs will support others in a given situation and each decision provides an opportunity to observe and refine the interactions. Secondly, in order to validate the interactions between KUs, it is necessary to derive predictions to be evaluated at later stages in the game. These would be predictions about the cards left in a player's hand, for instance. Subsequent invalidation, while minor in itself, could provide crucial evidence in a diagnosis of what went wrong when some major disaster struck (such as finding oneself inescapably in the lead at the end of the game in some suit in which every other player is void--and picking up hearts in every trick). So LH9 says that predictions should be made and evaluated. LH10 advises choosing a line of play which maximizes the feedback. Predictions are encoded as demons with triggers, such as "Player X does not have the queen of spades". If it turns out that Player X does have the queen of spades, and indeed that the system is forced to take it, then that demon would trigger. If someone else wins the queen, the system should not just forget to evaluate the support for that judgment, as people usually do. LH11 reminds the system to evaluate predictions which have not been explicitly triggered.

The other feedback heuristics impose some higher constraints on this process in order to make it effective. LH12 says simply that if a prediction has been made about a line of play, it is best not to let the other players disturb it. LH13 lies in the notion that there is a "horizon" of concepts which are sufficiently near to the current set to be reachable; reaching too far will probably diminish the chances of making progress. Its cautiousness is kept in check by LH14.

LH8. Apply all the knowledge all the time.
LH9. Make and evaluate predictions.
LH10. Follow the path generating the most predictions.
LH11. Collect and evaluate "expired" predictions.
LH12. Retain control as much as possible.
LH13. Don't try anything too difficult.
LH14. Don't do only things that are trivially easy.

Concept Generation:

Since we are viewing learning as a search through a space of concepts, there must be a way of generating concepts. The initial heuristic (LH15) states simply that new concepts should be formed from

the concepts that already exist. This heuristic may sound tautologous, but theories of learning have been proposed in which the KUs are formed simply by passively observing events. The remaining heuristics in this section are ways of guiding this process. LH16 through LH18 specify ways of using existing concepts which have been found to be reliable in use; they suggest generalizing, specializing, and composing them. LH19 suggests exploring semantically similar concepts when attempting to make an adaptive conceptual change. Unreliable concepts may be useful too if they contain a germ of truth. LH20 says that they may have to be split in order to derive reliable concepts. LH21 reminds the system that the current state variables constitute inputs to the concept generation process. LH22 advises that once information on an interaction has been obtained, it should be modeled as closely as possible (rather than introducing gratuitous generalizations at this stage).

LH15. Make new concepts by modifying old concepts.
LH16. Generalize reliable old concepts.
LH17. Specialize reliable old concepts.
LH18. Compose reliable old concepts.
LH19. Split unreliable concepts into component concepts, e.g., "danger" for high hearts versus for high clubs.
LH20. Explore the implications of concepts for semantically-related ones, e.g., compare high hearts with high spades, clubs, and diamonds
LH21. Incorporate aspects of the context. E.g., a high card is dangerous, but not if all others are void in that suit.
LH22. Modify concepts by most closely-modelled fix possible. E.g., evaluate "danger" by winners/losers not played rather than just number of cards not played yet.

Effectiveness:

The effectiveness of learning may be enhanced by modifying concepts only when a need to do so has been provided by the environment; LH23 codifies this. The effectiveness of concepts will need to be known for (a) choosing between them for application in a context, and (b) deciding which to retain when the space allotted to learning has run out and some poor concepts must be deleted to make space. LH24 suggests that accuracy of prediction is the most important factor in the effectiveness of concepts. LH25 gives a related criterion, that the most effective concepts are those in which the measure of accuracy is well-established; it suggests, in other words, that the variance is as important as the mean.

LH23. Modify concepts when motivated by a specific game-situation.
LH24. Prefer concepts which predict accurately.
LH25. Prefer concepts whose predictability has converged.

Efficiency:

One approach to efficiency in learning is to generalize whenever that is justifiable. Some techniques for doing this are set out in LH26 through LH29. LH26 suggests insuring that concepts have common subunits, and LH27 goes further in suggesting that attempts be made to massage groups of related concepts into one. One technique for doing these things is suggested by LH28; finding that the conditions of several concepts are related may allow the system to recognize that they are in fact just variations on one unifying concept. LH30 through LH32 suggest ways of making the overall process more efficient. LH30 suggests setting up demons so that once an element has been evaluated, it will only be re-evaluted when the situation changes enough for a re-evaluation to be necessary. This is related to the "common subunits" of LH26; the more subunits are shared between concepts, the less computation will go into evaluating each concept. LH31 suggest that generalization will facilitate these procedures.

LH26. Make common subunits where possible. E.g., "guaranteed loser."
LH27. Compact several concepts into one where possible.
LH28. Look for equivalence sets. E.g., a run of cards such as 6, 7, 8.
LH29. Organize concepts into implication chains where possible.
LH30. Compute predicates which are suspected to be true with special triggers which tell when they need to be re-evaluated.
LH31. Use a generalization of a concept for ease of computation and storage. E.g., which cards in each suit are guaranteed winners or losers rather than remembering each card played.
LH32. Discard over-specialized concepts which are little used.

Interpretation of others' behavior:

An important part of hearts is interpreting what the other players are trying to do, and analogies of this are also important in other learning situations. LH33 warns that this must be done with care; the system should not jump to conclusions. LH34 reminds the system to deduce all possible explanations of the player's actions, rather than just staying with the first found. LH35 makes the assertion that players are consistent. LH36 gives some advice on deducing other player's holdings. And LH37 gives the final caveat that another player will not always play the same way that the system will.

LH33. Deduce the other players' hands conservatively. E.g., discarding a low card may not mean the player has no high cards.
LH34. Deduce all strategies consistent with other players' actions.
LH35. Assume a player is consistent.
LH36. Remember there is a fixed quantity of cards. E.g.,

if neither you, nor players A and B have the queen of spades, then the other player must have it.

LH37. Do not assume that a player will necessarily play as you would.

Control of uncertainty:

One problem is that the Learner responds to specific situations, and these situations may be idiosyncratic in some ways. One counter to this is simply to have masses of experiences, but there are also ways of managing single experiences to extract the maximum of reliable information. LH38 proposes sticking with a given situation until the problem has been solved; attempt after attempt will allow refinement of the concepts even if success is not attained. LH39 stipulates that even successful plays should be repeated, with minor variations. This will allow the Learner to detect and characterize boundary conditions. Following the principles of common subunits and demons enunciated above, such further experiences will be computationally cheaper than simply waiting for appropriate occasions to present themselves for exploring the bounds on concepts. LH40 is advice on remembering the sequence of events that led up to the situation; instantiated for the hearts game, it says to remember the sequence of cards played. And LH41 suggests an overall control for uncertainty--to remain flexible.

LH38. Repeat the same situation until it is mastered.
LH39. Try variations on the plays.
LH40. Remember which cards have been played as far as possible.
LH41. Remain flexible.

Conflict Resolution:

Often, two or more avenues may remain open after an analysis of the situation. LH42 through LH46 offer concrete suggestions about how to evaluate alternatives. The first says that a sacrifice may have long-term gains. Moreover, those anticipated gains will be set out explicitly as demons, and if they are not achieved, a productive game analysis will be instigated. LH43 is related; given that one wants to win, one should prefer the path which has expectations extending furthest into the future, subject to a reliability constraint. LH44 suggests diversification; there is more to be learned from a decision-point if more knowledge can be brought to bear upon it. And LH45 is the converse of LH43--when the game is not in danger a risky path may be followed because it has more wide-reaching implications for the evaluation of concepts.

LH42. Do a low cost thing now to stop a high cost thing later.
LH43. Follow the longest most certain path, when the situation is difficult.
LH44. Increase your options in any way possible.
LH45. Do what will teach you more, when the situation is easy.

Appendix VIII: RULES OF HEARTS AND SAMPLE INITIAL HEURISTICS SET

Hearts Rules:

1. There are three or four players. Everyone plays against everyone else.

2. The object of the game is to get as few points as possible.

3. Each heart is worth 1 point. The queen of spades is worth 13 points.

4. A game usually terminates when one player has accumulated 100 points.

5. Each round of the game starts with a deal. If there are four players, each player will get 13 cards; if there are three players, each will get 17, with one left over.

6. Next, each player selects 3 cards of his to pass to another player. With 3 players, the pass-direction is alternately clockwise and anti-clockwise; with 4, it may go in one direction for the whole game, or alternate between these two and passing to the player opposite. Naturally, no player may see what cards are being passed to him before choosing the cards he will pass.

7. The player with the lowest club leads the first trick. A trick consists of each player playing a card, one after the other in a clockwise order.

8. Each player must play a card in the suit led if he has any in that suit; otherwise, any card may be played (this is called "sluffing" a card when you're "void" in a suit).

9. Hearts may not be led until either the queen of spades or a heart has been played, unless the player leading has only hearts left.

10. The player playing the highest card in the suit led wins the trick and is charged any points the trick contains. If there are only three players, the winner of the first trick takes the extra card left after the deal. No one else sees this card.

11. If a player takes all 26 points in a round, the other players are all charged 26 points. This is referred to as "shooting the moon."

Sample Initial Heuristics for Playing Hearts:

H1. If you're leading and going low and have enough spades to last without getting the queen of spades, then lead a spade below the queen.

H2. Any card that helps you void is a good card.

H3. Any card that can cause you to get the queen of spades is a bad card.

H4. Any card that is a guaranteed loser is a good card.

H5. Getting rid of guaranteed losers is undesirable unless necessary.

H6. Leading a low Heart is good.

H7. Playing a winning card is good if the threat is low.

H8. If player X is shooting the moon and player Y might win the trick, then play a heart if possible.

H9. If player X is shooting the moon and there is a heart in a trick you can take, then take it.

H10. If player X is shooting the moon and there's no heart yet taken against him, and you can take the trick and you can take subsequent tricks with a high chance of getting a heart, do that.

H11. If someone's shooting the moon and you can't win the trick, don't play any hearts that eliminate all potential winning hearts (i.e., don't sluff winners).

H12. If you have a loser in a suit where some but not all opponents are void, play the highest loser.

H13. If you're void in the suit led and you're not trying to prevent others from shooting the moon, play the queen of spades or your highest heart.

Appendix IX: CRITERIA FOR A WELL-FORMED KNOWLEDGE BASE FOR INTELLIGENT AGENTS

Our goal is to design and implement new methods for developing intelligent agents. We feel this has general applicability for knowledge-based systems of all kinds. Our examples have been drawn from the domain of Hearts which has served as a good testbed for our ideas. We describe here what we feel are essential criteria for the general problem of designing and organizing a knowledge base for use by intelligent agents, particularly those whose performance will be improved over time.

The criteria we identify are:

1. Knowledge should be inspectable.
2. Knowledge should be represented uniformly.
3. Knowledge should be extensible.
4. Knowledge should be connected.

We discuss each in turn.

Inspectability

For an IA to analyze its own behavior, to explain its processing and results, and to support extensibility, knowledge must be transparent, explicit, and accessible. In short, knowledge should be coded in an inspectable formalism. For this reason, we feel that knowledge representations should be primarily declarative rather than procedural. In conventional IA programs, knowledge frequently lies hidden in some procedure or lambda expression not readily amenable to analysis and modification. Our own experience has shown how frustrating and time-consuming changing even trivial amounts of knowledge can be when it is embedded in executable code. Our experiments with implementing various control structures support this viewpoint (see also Appendix III). To change from a top-down control structure to a hypothesize-and-test paradigm, we had to recode the (procedurally represented) control mechanism. The heuristics, on the other hand, had been represented declaratively (in the form of concepts) and required little or no modification to support the new control scheme.

Learning necessitates explicit knowledge representations to identify which KUs participte in system behaviors of interest. Improved performance may require abilities to look at and reason about conceptual definitions, heuristics, control structures, evaluators, functions, etc. The Learner should be able to examine all facets of the decisionmaking process. It must be able to locate (in fact, pinpoint) good and bad knowledge units, as well as decide where and how new knowledge should be inserted and used. Conversely, opaque or inexplicit knowledge representations create requirements for human programmers to intervene whenever knowledge needs to be changed, updated, or extended. Our ultimate objective is to automate as much of this process as possible.

Uniformity

To support efficient and general inference procedures, a uniformly represented knowledge base seems desirable. Organizing and integrating knowledge are made increasingly tedious and time-consuming when diverse forms of knowledge are represented differently. Methods for retrieval, insertion, evaluation, and explanation should operate over a uniform representation. Otherwise alternative methods for these operations must be developed for different representations.

We are not arguing for uniform representation of input (advice) to the system. We want to process advice that is informal and imprecise. However, we believe we can and should translate varying external descriptions of knowledge units into uniform internal representations with well-defined semantics. It is the function of the Knowledge Programmer to map the informal inputs into such uniform representations. To the extent possible, the uniform representation should be simple, understandable and modifiable.

Extensibility

Rapid knowledge base construction begins with the formulation of primitive knowledge units. A primitive KU is one which references predefined constants, literals and functions. Subsequent knowledge should be built upon these primitive units or, recursively, upon other KUs extant in the knowledge base. Thus we view knowledge engineering as being a process that is incremental and additive. We do not require that knowledge input to the system must follow these hierarchical structuring principles explicitly. However, the Knowledge Programmer must be able to impose such structure upon its interpretations of the informal advice. This criterion relates to the following connectedness principle.

Connectedness

If a knowledge unit were conceptually isolated, it would be of no value. KUs should always be connected to other KUs. In this context, two KUs connect whenever one produces or consumes data required by the other. For example, planning heuristics often connect to lower-order heuristics that refine their intentions or methods; heuristics connect to the control structures that apply and integrate them; functions, predicates, and variables connect to heuristics and control structures that use them; heuristics connect to plans, goals, objectives; meta-heuristics prescribe good ways to integrate, evaluate, and control other knowledge, etc. New knowledge, introduced externally by an expert or internally by the Learner, must connect to existing KUs to be meaningful and usable.

Connectivity has dynamic aspects of importance also. Learning requires an ability to monitor and interpret the data flow of knowedge-based inferences. These functions help identify which KUs warrant modification. Similarly, general explanation capabilities would identify and then describe knowedge flows with the system that led to events of interest.

PREFACE

This paper presents the technical portion of a recent proposal to the Intelligent Systems program of the National Science Foundation. It proposes a new paradigm for building knowledge-based systems and a specific research project emphasizing knowledge acquisition and machine-aided knowledge programming. Many of the ideas described should be of interest to researchers in artificial intelligence and cognitive psychology. Like most proposals, however, this one contains many incompletely developed concepts. As a consequence, the interested reader may discover as many questions as answers suggested by this paper.

$5.00